TU WIEN Informatics

Geometry-based Railway Track Extraction from OpenStreetMap Data

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Matthias Eder, BSc
Matrikelnummer 01633017

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Inform. Dr.rer.nat. Martin Nöllenburg
Mitwirkung: Dr.techn. Dipl.-Ing. Markus Wallinger
Dr.techn. Dipl.-Ing. Soeren Terziadis

Wien, 3. Mai 2024

Matthias Eder	Martin Nöllenburg

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Geometry-based Railway Track Extraction from OpenStreetMap Data

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Matthias Eder, BSc
Registration Number 01633017

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Inform. Dr.rer.nat. Martin Nöllenburg
Assistance: Dr.techn. Dipl.-Ing. Markus Wallinger
Dr.techn. Dipl.-Ing. Soeren Terziadis

Vienna, 3rd May, 2024

Matthias Eder	Martin Nöllenburg

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Matthias Eder, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 3. Mai 2024

Matthias Eder

Acknowledgements

I extend my heartfelt gratitude to my supervisors Prof. Martin Nöllenburg, Dr. Markus Wallinger and Dr. Soeren Terziadis for their guidance and support throughout the thesis. Their expertise, invaluable feedback, suggestions and encouraging words in our countless meetings were indispensable for the creation of this thesis. I would also like to thank my colleagues, Dr. Senen Gonzalez and Dipl.-Ing. Leonhard Kreil-Brunauer from tmc, for originating the idea for this thesis and for providing me with the necessary infrastructure, insights and support on my journey.

Kurzfassung

Geografische Datenbanken stellen meist lineare Elemente wie Flüsse, Straßen und Eisenbahnstrecken als *Polylines*, Sequenzen von Punkten, die diese Elemente approximieren, dar. In Datensätzen wie beispielsweise von OpenStreetMap (OSM), werden Eisenbahnstrecken als ungeordnete Mengen von Polylines gespeichert. Dies kann für Anwendungen, die eine sequenzielle Verarbeitung dieser Merkmale erfordern, in weiterer Folge zu Herausforderungen führen. Diese Diplomarbeit behandelt das Problem der Transformation ungeordneter Mengen von Streckensegmenten in eine einzelne Lösungsmenge. Die aus einem Algorithmus resultierende Lösungsmenge soll alle Eisenbahnstrecken aus der Eingabemenge darstellen, wobei jedes Element der Lösungsmenge eine durchgehende Polyline ist. Der Algorithmus soll ganze Eisenbahnstrecken möglichst genau aus der Eingabemenge erkennen, die Eisenbahnstrecken der realen Welt akkurat repräsentieren und dabei möglichst stabil gegen Bearbeitungsfehler sein. Wir definieren einen Prozess, der Streckensegmente in einem zweidimensionalen Koordinatensystem ordnet und verbindet, um dies zu bewerkstelligen.

Angestoßen durch Anforderungen der Track Machines Connected GmbH (tmc), einem Unternehmen für Eisenbahndigitalisierung, für eines ihrer Software-Projekte, präsentiert diese Arbeit einen rein geometriebasierten Ansatz zur Lösung des Problems. Die vorgestellten heuristischen Algorithmen nutzen Strategien wie das Verbinden von Streckensegmenten und eine Approximierung der Wahrscheinlichkeiten von Streckensegmentverbindungen. Dabei wird das Problem auf ein Graphenproblem, das *Maximum Weighted Path Cover Problem*, reduziert und gelöst.

Zu den behandelten Themen und Schlüsselfragen gehören die Charakterisierung problematischer Eingabe-Instanzen, die Beschreibung und Darstellung der algorithmischen Verfahren und ein Vergleich der Algorithmen mit unterschiedlichen Parametern, hinsichtlich Lösungsqualität und Laufzeiteffizenz. Die Ergebnisse dieser Arbeit zeigen, dass der hier präsentierte heuristische Algorithmus eine signifikante Verbesserung gegenüber einer einfachen Proof-of-Concept Lösung darstellt.

Abstract

Geographic databases commonly represent linear features like rivers, roads, and railway tracks as *polylines*, which are sequences of points approximating these features. However, in datasets like OpenStreetMap (OSM), railway tracks are stored as unordered sets of polylines, posing challenges for applications requiring sequential processing of these features. This thesis addresses the problem of transforming unordered sets of track segments into a single set of polylines, the solution set, to accurately represent real-world railway tracks while being robust to editing errors.

We define a process to connect and order track segments in a two-dimensional coordinate system to accurately represent railway systems. Motivated by the needs of Track Machines Connected GmbH (tmc), a railway digitalization company, this thesis proposes a geometry-based approach to solve the problem. The introduced heuristic algorithms utilize strategies such as track segment merging and defining weight functions to determine the probability of segment connections. A reduction of this problem to a graph problem, the *Maximum Weighted Path Cover Problem*, is used in our approach to solve the problem.

Key questions addressed include characterizing problematic input instances, defining the algorithmic methods, metrics for solution quality, and performing an algorithm comparison of the algorithms, with different parameters, in terms of solution quality and runtime performance. The results of this work demonstrate that the heuristic algorithm presented in this thesis constitutes a significant improvement over a simple proof-of-concept solution.

Contents

CHAPTER 1

Introduction

In most geographic databases linear features such as rivers, roads or railway tracks are represented as a set of *polylines*. A polyline (also called *linestring*) is a curve defined by a sequence of points that approximates the feature. In OpenStreetMap (OSM) [20], a project for collecting and structuring geodata that provides a database with open access, railway tracks are represented as a set of polylines which are not necessarily ordered. For cases such as rendering tracks, the order of polylines is not important. However, simply fetching the data, an unordered set of polylines, from OSM might not always be sufficient. When, for example, processing in sequence over real-life geometry is required, it is advantageous to represent geospatial features as a single polyline. Transforming this kind of geospatial data to a single polyline or a set of polylines that represent a real world path or track accurately turns out to be a challenging problem. This can, for example, be problematic if a linear feature is self intersecting, meaning its shape contains a loop. Figure 1.1 shows an example of a self intersecting track. The red parts show segment ends where the next segment in the order cannot be deduced easily, and multiple possible solutions are feasible. In the case of Figure 1.1 the top right shows the solution which represents the real world geometry more accurately than the suboptimal solution in the bottom right. For railway tracks in alpine regions, this is a common problem which can occur on tracks in mountains. For example, if a track winds up a mountain, one section of the track could lead through a tunnel while a later section lies directly on top of it on the same geographic coordinates. Additionally, there can be small inaccuracies in the exact coordinate locations, due to editing errors or measuring inaccuracies. The OSM database consists of a huge amount of data collected and maintained by a large amount of contributors around the world. Inevitably, at some point inconsistencies in the data or data format will appear. Since these polylines are defined in a two-dimensional-space, there is no easy way to determine whether two points in the coordinate system have the same altitude.

The goal of this thesis is to find an algorithm that transforms unordered sets of track

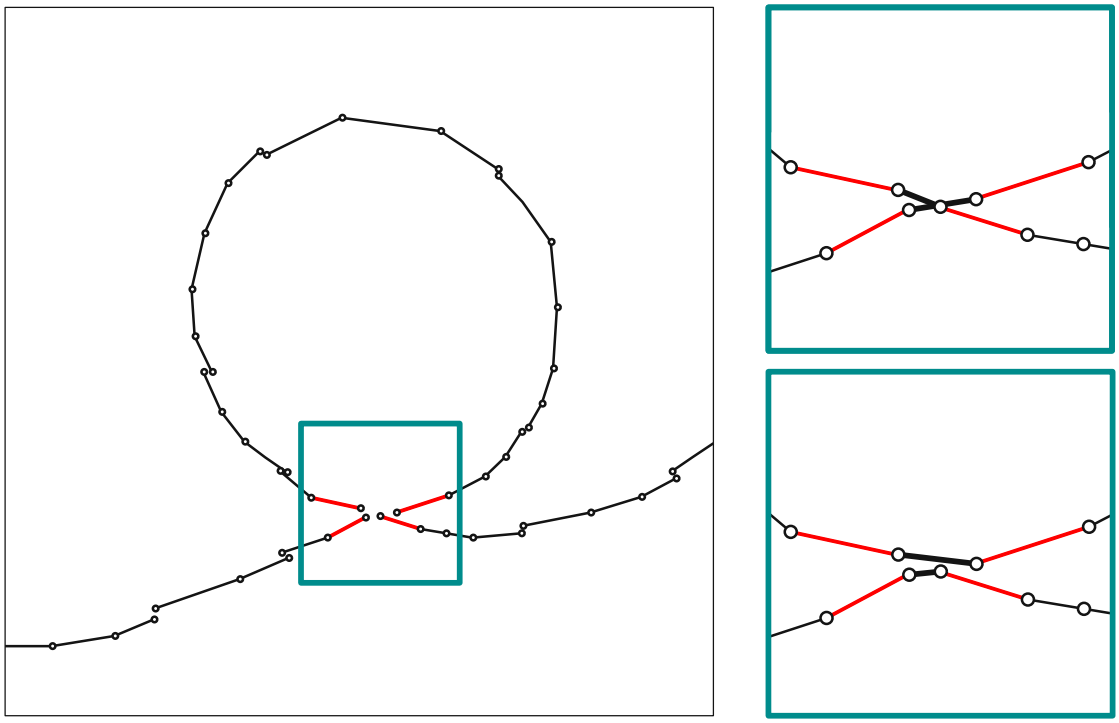


Figure 1.1: Example of ambiguous track segments (left) and possible solutions (right).

segments into ordered ones. The output should represent real world railway tracks as close as possible, while being robust to editing errors.

Formally one can define the abstract research problem as:

"Given a set of unordered track segments, where each track segment is defined by a sequence of points in a two-dimensional coordinate system. Find a process to connect and order those segments such that the connected segments accurately represent the rails in a railway system map"

The idea for this thesis originated from the Track Machines Connected (tmc) [1], which is a railway digitalization company that needed a solution to this problem for one of their software projects, TrackDB.

In this thesis we propose a geometry-based approach to solve this problem. The algorithms we introduce use several strategies, like merging segments that are unambiguously connected to each other in a pre-processing step or using weight functions for a heuristic approach to determine the probability of two segments to being connected. The findings and outcome of this thesis have practical implications within the TrackDB software, while also holding potential applicability across other infrastructure networks like motorways or pipeline systems.

We address the following key questions in this thesis:

1. How can we characterize problematic input instances and locate subsets of segments with ambiguous order which might lead to accuracy problems?
2. How do the suggested algorithms improve the current approach to the problem?
3. What are suitable measures of the solution quality of the algorithms?
4. How do the presented algorithms compare to each other in terms of solution quality and runtime performance?

This thesis is structured as follows: In Chapter 2 we present related work to the topic of this thesis, that has been published by other authors. In Chapter 3 we introduce several definitions and information that will be needed for the further understanding of the following chapters. In Chapter 4 we discuss an existing proof of concept solution to the problem that has already been developed, describe the structure and discuss its shortcomings. This approach sets the fundamental base of the algorithm we design and introduce in Chapter 5 and Chapter 6. Chapter 5 shows one way of fetching raw OSM railway data and converting it into the data structures used by our algorithm. Further we show in this chapter how the data structure is constructed. Chapter 6 describes the process of finding long paths, possible solutions, using the data structure from Chapter 5 on given input instances. We performed several experiments to give sample results of our algorithm on multiple input instances and evaluate the performance and solution quality of the algorithm in Chapter 7. Chapter 8 concludes this research and gives an outlook on possible future work.

CHAPTER 2

Related Work

In this chapter, we explore existing literature relevant to our methodology for extracting railway track geometry from OpenStreetMap (OSM) data. Our approach involves heuristic mapping of input data to a graph structure and reducing the problem to a maximum weighted path cover problem. The vertices of this graph structure represent track segments and the weights of edges of adjacent vertices the likelihood of them being geometrically connected. The mentioned concepts will be discussed in more detail in the following sections. Therefore, research on related work is focused on three main topics: Representing OSM data in a suitable graph structure, addressing the maximum weighted path cover problem and finding an appropriate weight function for our heuristic.

2.1 Representing OSM Data in a Graph Structure

The processing of geospatial data from geographic databases like OSM is a relatively widely researched topic. There are several projects that transform OSM data into other formats like graph structures. OSMnx [2] is a tool which provides options to collect data from OSM, create street networks and perform analysis on them. It includes functions like calculating the shortest path between two points. The tool, however, does not cover the problem we tackle in this thesis, computing polyline geometries from (maybe inconsistent or ambiguous) data to depict railway tracks, as OSMnx focuses on transforming street network data to a graph structure simplifying and enabling faster processing of the data.

Rahmig and Kluge [23] explain the OSM railway data structure and present an extension to the OSM data structure which tries to fulfill a vast variety of application requirements and aims to allow for an easier modelling of digital maps. The introduced data structure is a graph where vertices represent locations at which tracks are diverging, branching, or crossing each other and edges represent the tracks between those vertices.

A protocol to convert spatial polyline data to network formats is presented in the paper by Karduni et al. [12]. In this protocol road crossings, bridges and tunnels are mapped to vertices and the road segments in-between to edges.

Although Rahmig and Kluge's graph structure [23] as well as the one presented by Karduni et al. [12] are proficient for traversal and pathfinding purposes, they lack the inclusion of track segment geometries and properties within the graph's vertices.

Chong et al. introduce a track segment graph for multiple hypothesis tracking in [5]. The track segment graph from [5] finds application in object tracking. Track hypotheses are assigned a score based on track segment association likelihoods which closely relates to the situation we want to represent in our graph structure.

One particular problem that inevitably leads to issues converting OSM data to tracks and generating a correct graph structure are errors in the data fetched from OSM itself. There are existing tools for detecting invalid OSM data like JOSM/Validator [11] or OSM Inspector [7]. JOSM/Validator is able to detect duplicated or incomplete ways and OSM Inspector is able to show basic geometry problems such as overlong ways. The strategy for extraction from OSM data we propose in this paper should be able to run automatized, thus we need to make our algorithm robust to such errors. The paper of Tabes et al. [26] proposes an algorithm fixing the connectivity in unconnected street networks, an issue that can arise with faulty OSM data. After converting the street network to a graph structure and detecting two unconnected portions of the graph, the algorithm tries to find the best way to create a strongly connected component including both subgraphs by trying to add the least amount of nodes to the component.

2.2 Path Cover Problem

One approach that we briefly mentioned there is a reduction of the original problem to a maximum path cover problem. Moran et al. [19] and Kobayashi et al. [13] are of particular interest for solving the maximum weighted path cover problem. In Moran et al. [19] three different approximation algorithms for the maximum path cover problem are presented. The paper describes 2/3-approximation algorithms for undirected graphs and directed graphs.

Chen and Ravichandran [4] present a maximum weight constrained path cover algorithm for graph-based multitarget tracking based on the track segment graph from Chong [5] that we briefly discussed in Section 2.1.

As mentioned by Lin and Ren in [15] the maximum weighted path cover problem is NP-complete. This complexity classification of the problem suggests that a heuristic approach to this problem is a reasonable choice.

2.3 Track Geometry Considerations

In order to define an appropriate weight function for our heuristic, detailed information about the standardized layout of railway tracks is advantageous. Legal regulations for the railway track infrastructure in Austria are defined on the official "Rechtsinformationssystem der Republik Österreich" (RIS) website [25] from the Austrian government. The thesis "Localization of Trains and Mapping of Railway Tracks" [9] describes the required properties of railway tracks from a physical and legal perspective in more detail. Those properties include, for example, the maximum curvature, or the continuity of the course of the track. They will be used to define the weight function for our heuristic. Sellerhof and Berkhahn [27] present an abstract method to model the layout of railway lines in a 3D-space with b-splines. Unfortunately, not that much literature can be found for the problem of merging and ordering polylines to retrieve a single polyline in itself. The paper [28] of Wolin et. al. presents an algorithm to detect corners in hand-sketched strokes. Some methods used in the algorithm from this paper could be used in an altered way for track segments.

CHAPTER 3

Preliminaries

3.1 Mathematical Definitions

In this section we define geometric terms and concepts that are used in the following chapters.

Basic geometries

A *point* P in $\mathbb{R}^2$ is an object that specifies a location in a two-dimensional space. It is defined by an x- and y-coordinate.

A *curve* C is a continuous mapping from an interval of the real numbers to a space. In plane $\mathbb{R}^2$ this mapping is defined as $\phi : [0,1] \to \mathbb{R}^2$. We say a point P lies on a curve if there exists a $t \in [0,1]$ such that $\phi(t) = P$. One can split a curve C into two *curve segments* by defining a *join point* $P = \phi(t)$ where $t \in [0,1]$. The two curve segments are then defined as $\phi_1 : [0,t] \to \mathbb{R}^2$ and $\phi_2 : [t,1] \to \mathbb{R}^2$. Continuity $\mathcal{C}$ describes how two segments of a curve C come together at a point (also referred to as *smoothness*). At a given join point P the continuity is defined as:

- $\mathcal{C}^0$ **continuity:** if the two curve segments share the same point where they join: $\phi_1(t) = \phi_2(t)$.
- $\mathcal{C}^1$ **continuity:** if the two curve segments are $\mathcal{C}^0$ continuous and have the same tangent (share the same parametric derivatives) at the join point: $\phi_1'(t) = \phi_2'(t)$.
- $\mathcal{C}^2$ **continuity:** if the two curve segments are $\mathcal{C}^1$ continuous and have the same curvature (share the same second parametric derivatives) at the join point: $\phi_1''(t) = \phi_2''(t)$.

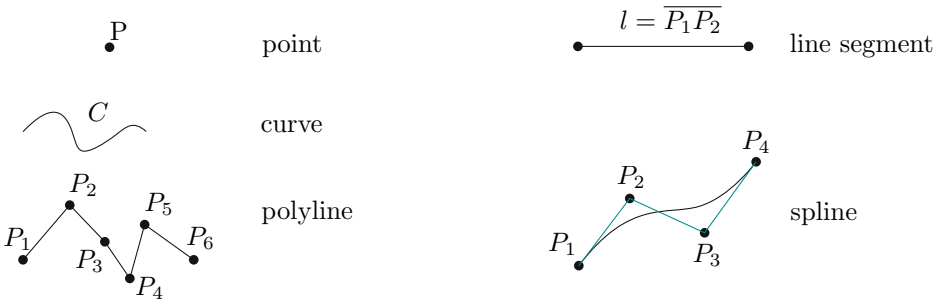


Figure 3.1: Basic geometries in $\mathbb{R}^2$

If the curve has $\mathcal{C}^n$ continuity (share the same 0'th ... n'th parametric derivatives) for any arbitrary join point P, then the curve is said to be $\mathcal{C}^n$ continuous.

Let $P_1 \dots P_i$ be points in $\mathbb{R}^2$. A *polyline* or *line string* is defined by an ordered sequence of such points $P_1 \dots P_i$. Given two points P and Q in $\mathbb{R}^2$, a *line segment* $l = \overline{PQ}$ from P to Q is defined as the point set $\{P + t * (Q - P) \mid t \in [0, 1]\}$. In the course of this thesis we will use the term *track segment* or short *segment* as an abstract term describing the geometry of sections of a railway track. A track segment is a polyline $P_1 \dots P_i$ where $i \geq 2$ (a polyline with at least two points).

A *spline* is a parametric curve defined by control points. The *control points* of a spline are a finite set of points that define the shape of the spline. Control points do not necessarily have to lie on the spline itself. There are different types of splines, such as *Bézier splines*, *Hermite splines*, and *B-splines*.

Line segments, polylines and splines are all also curves by definition.

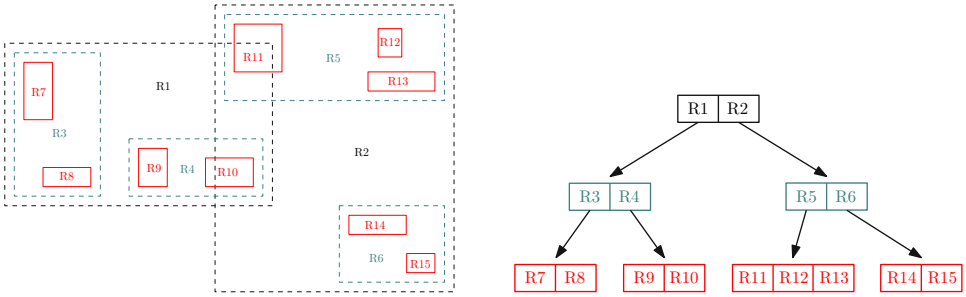
Graphs

A *graph* $G = (V, E)$ is a pair of a set of vertices V and a set of edges E. In an *undirected graph*, an *edge* $e_{i,j}$ is a 2-element subset of V where $E \subseteq \{\{v_i, v_j\} \mid v_i, v_j \in V\}$. Whereas in a *directed graph*, each edge is an ordered pair of vertices where $E \subseteq \{(v_i, v_j) \mid v_i, v_j \in V\}$.

Two vertices v_i and v_j are called *adjacent* or *neighbors* if there is an edge $e_{i,j}$ between them (i.e. $\{v_i, v_j\} \in E$ for undirected graphs or $(v_i, v_j) \in E$ for directed graphs).

A *weighted graph* is a graph where each edge has an associated weight ω. The weight for each edge $e_{i,j}$ in a weighted graph is denoted as $\omega_{i,j} \to R_{0+}$.

A *path* P in a graph is a sequence of vertices $(v_1, \dots v_n)$ with $n > 1$ where each vertex is adjacent to the next vertex in the sequence. Vertices in a path are pairwise distinct and there exist $n - 1$ edges $e_{1,2} \dots e_{n-1,n} \in E$ in the graph where $e_{i,i+1} = \{v_i, v_{i+1}\}$ for undirected graphs and $e_{i,i+1} = (v_i, v_{i+1})$ for all i with $1 \leq i < n$. The *length* of a path P is defined by the number of its vertices minus one. Two paths P_1 and P_2 are *vertex-disjoint* (or short *disjoint*) if they do not share any vertices. For two disjoint paths P_1 and P_2 for each vertex $v_i \in P_1 \implies v_i \notin P_2$ must hold.



(a) Search Space (b) R-tree

Figure 3.2: Two-Dimensional R-tree Example

A *bounding box* is a closed region that completely contains a set of geometric objects. For a single object in $\mathbb{R}^2$ we define (if not specified otherwise) the bounding box of said object as an axis aligned rectangular bounding box of minimal size.

R-tree

An *R-tree* is a dynamic index structure for spatial searching holding its indexed records in the leaf nodes. It is height-balanced similar to a B-tree and organizes its objects as intervals in multiple dimensions. Each node spans an n-dimensional area defined by n intervals. All child nodes are spatially contained in the area of their parent node meaning that every interval of the child node is a subset of the parent's interval for all n dimensions. Upon searching for records that overlap a queried search area the nodes of the R-tree are traversed from top to bottom (i.e. from root to leaf nodes). Figure 3.2 shows an example of a two-dimensional R-tree with Figure 3.2a being the search space containing search intervals depicted in dashed rectangles and the indexed records in red rectangles. In Figure 3.2b the corresponding R-tree can be seen. Only nodes that overlap the search area are traversed further, thus allowing for a fast search. A more detailed description of the R-trees and algorithms for inserting nodes, updating nodes and finding leaf nodes can be found in "R-trees: dynamic index structure for spatial searching" [8] by Guttman.

3.2 Representing Track Geometries in OSM

The biggest advantage of using OSM as a data source opposed to other data providers is the huge amount of data that is available. Usually, railway data providers such as construction companies or railway operators only manage data of their own or a just small amount of tracks and sometimes do not provide their data openly. Since OSM is crowdsourced, there is data almost everywhere in the world. Specifications and feature descriptions of OSM data can be found in "OpenStreetMap Data in Layered GIS Format" [24]. One drawback with OSM, however, is that the data is very coarse and might not always be that accurate. Additionally, the data from OSM does not contain information

about the altitude of a track position or sometimes lacks other detailed information such as track kilometers, railway identification number and others. Therefore, we need to estimate the course of the tracks solely based on the geometry data of the segments. They contain the start and end point of the segment and the control points in between as coordinates in a two-dimensional space. As described in [25] and [9], the course of a railway track has to fulfill certain requirements per law. These requirements include the maximum curvature of a track, the continuity of the course of the track, maximum speed and others. For example, the minimum curve radius is 100 meters and the track needs to have a $\mathcal{C}^2$ continuity. We can take advantage of the knowledge of these requirements to define a proper heuristic for the problem.

The process of constructing single railway lines from OpenStreetMap data can be divided into multiple subproblems. The subproblems range from well explored topics, like the extraction of raw data from OpenStreetMap, to more specific problems like the ordering of segments to construct solutions.

CHAPTER 4

Greedy Local Fit Algorithm

Remember the definition of the abstract problem: *"Given a set of unordered track segments, where each track segment is defined by a sequence of points in a two-dimensional-space coordinate system. Find a process to connect and order those segments such that the connected segments accurately represent the rails in a railway system map"*

Track segments are polylines which contain an ordered finite set of points (and hold additional metadata). The *track creation algorithm* creates a set of tracks from segments where each element in the result set should represent a single railway track. Each track in the computed solution is an ordered set of polylines.

A proof of concept implementation for this problem has already been built by TMC. Figure 4.1 shows the processing pipeline of creating tracks from raw OSM data. The process can be divided into four steps:

1. Data Extraction
2. Track Creation
3. Post Processing
4. Output Result

In this section we will give an overview over the proof-of-concept solution. Chapter 5 focuses on Step 1, the extraction of data from OSM. Step 2, the core part of the track creation, will be covered in Chapter 6 where we introduce our new approaches. Steps 3 and 4 are not essential parts of this thesis and will only be briefly discussed in this Chapter and Chapter 7.

OSM data is fetched using the Overpass API, where tracks are selected by a given railway reference number. The reference number needs to be present in the OSM Tags.

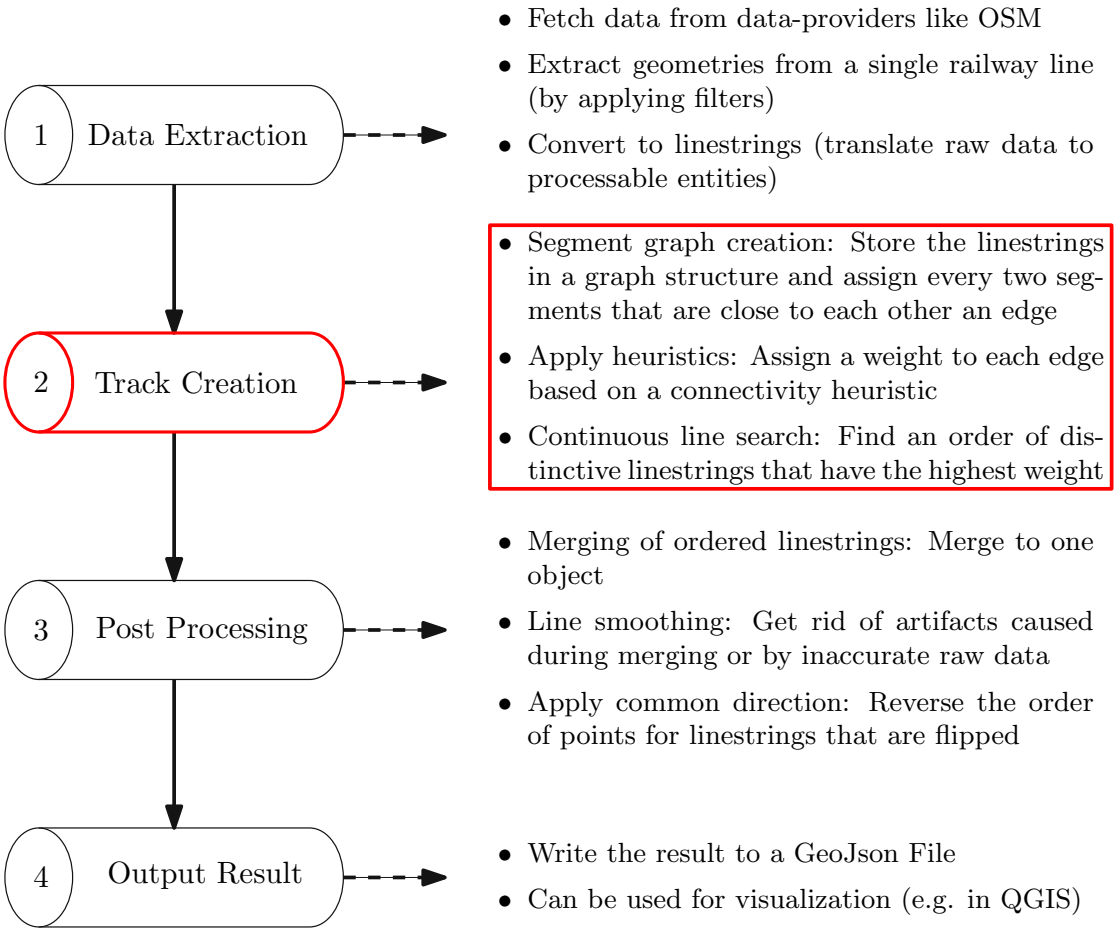


Figure 4.1: Processing pipeline of existing solution

Unfortunately, railway tags are not always present or accurate, especially outside of Europe. The primary benefit of this approach is that it enables the retrieval of only those track segments associated with the specified railway reference number. For the track creation a graph data structure is used to hold the segments. The graph structure for segments is described in more detail in Section 4.1. In order to add a scoring on different combinations of orders of segments, a weight function defines a weight for each pair of segments. Segments which are not in the neighborhood of each other are omitted. In Section 4.2 the weight function is formally defined. After the creation of the segment graph and the assignment of all weights, the algorithm tries to find continuous lines by starting at a random segment and following the path with the highest weight. The algorithm stops when no more segments can be added to the path. For the expansion of each part solely the weight of the edge to the next neighboring segment is considered. After the paths have been found, segments that are part of the same path get merged together in the same order to get a single linestring for each path. Some smoothing and other post-processing operations are then applied to the continuous lines to improve the

result. Subsequently, the results are converted back to GeoJson and written to a file which can then be used for further processing.

4.1 Segment Graph

To solve the problem, TrackDB currently employs an approach that reduces the general problem to a graph problem. Segments are mapped to vertices in a corresponding graph structure that we call the *segment graph*. The edges of the segment graph and their weights are associated with the probabilities of the vertices (segments) to be connected in the real world.

We define $G_{seg} = (V, E)$ as a *segment graph* with vertices V and edges E where V is a set of track segments and E a set of potential connections. The expression $e_{i,j}$ represents a potential connection between track segments v_i and v_j. Additionally, we define $\omega(e_{i,j})$ or short $\omega_{i,j}$ as the weight of the edge $e_{i,j}$. The weight $\omega_{i,j}$ on an edge $e_{i,j}$ in a segment graph is determined by a weight function ω for every potential connection of two track segments. $(v_1, v_2, \ldots, v_n)$ is a sequence of track segments, where v_1 is the first element followed by v_2 as its *successor* and v_n as the last element. v_1 is the *predecessor* of v_2 accordingly. A successor track segment and its predecessor are called neighboring track segments or *neighbors*. That implies v_1 is *neighbor* of v_2 and v_2 is neighbor of v_1. The *delta value* δ is a threshold that defines the maximum distance that two segments v_1 and v_2 can be apart in order to be neighbors:

$$v_1 \in neighbors(v_2) \iff distance(v_1, v_2) \leq \delta$$

and

$$v_2 \in neighbors(v_1) \iff distance(v_1, v_2) \leq \delta$$

Figure 4.2 shows a simple representation of the currently used data structure in the algorithm: where $v_1, \ldots, v_i$ are the vertices in the segment graph which represent the segments. Segments are stored as bounding boxes in an R-tree, which is used for indexing and determining the neighborhoods of segments. Note that these bounding boxes are spanned over just the start and the end point. So points in a segment between start and end point can lie outside the bounding box but are of no further interest. A segment v_j is in the neighborhood of v_i if the spatial distance between v_i and v_j is lower than a given threshold δ. The value of δ is variable and set to 1 meter by default. The expression $e_{i,j}$ denotes an edge from v_i to v_j in the segment graph where, based on a weight function, a weight ω can be assigned to each edge. The weight $\omega_{i,j}$ represents the probability that two vertices (segments) v_i and v_j are connected. Edges between segments which are not in the same neighborhood, will be discarded from the segment graph. Algorithm 4.1 shows the algorithm for creating the segment graph in pseudocode. For every segment the algorithm checks for neighboring segments and adds an edge between them if they are not already connected. First, neighboring segments from the start point of the current segment are checked and later from the end point. For a candidate v_j (line 3, 4)

Algorithm 4.1: Algorithm for creating the segment graph

Input : *segments*: A set of polylines with their metadata
Result: *segmentGraph*: $G_{seg} = (V, E)$

1 $V \leftarrow segments$;
2 $E \leftarrow \emptyset$;
3 **foreach** $v_i \in V$ **do**
4 $\quad predecessorCandidates \leftarrow \{v_j \mid v_i \in V \wedge distance((v_i.startPoint), v_j) < \delta \wedge v_j \neq v_i\}$;
$\quad$ /* All track segments that lie in the radius of δ from the start point (the neighbors of segment v_i) */
5 $\quad succesorCandidates \leftarrow \{v_j \mid v_i \in V \wedge distance((v_i.endPoint), v_j) < \delta \wedge v_j \neq v_i\}$;
$\quad$ /* All track segments that lie in the radius of δ from the end point (the neighbors of segment v_i) */
6 $\quad$ addEdges(v_i, *succesorCandidates*);
7 $\quad$ addEdges(v_i, *predecessorCandidates*);
8 **end**
9 $G_{seg} \leftarrow (V, E)$;
10 **return** G_{seg}

11 **Function** addEdges(v_i, *candidates*)**:**
12 $\quad$ **foreach** $v_c \in candidates$ **do**
13 $\quad\quad$ **if** $e_{i,c} \notin E$ **then**
14 $\quad\quad\quad E \leftarrow E \cup \{e_{c,i}, e_{i,c}\}$; /* Add edge in both directions and determine weights */
15 $\quad\quad\quad \omega_{c,i} \leftarrow weight(v_c, v_i)$;
16 $\quad\quad\quad \omega_{i,c} \leftarrow weight(v_i, v_c)$;
$\quad\quad\quad$ // $weight = (distance * angle * intersection)$ between v_i and v_j
17 $\quad\quad$ **end**
18 $\quad$ **end**

the current algorithm always checks first whether the candidate's start point is in the neighborhood of the current segment v_i. Only if it is not, then the end position of v_j is checked. This coordinate of v_j is saved and used for the weight function's distance check. When evaluating the weight for a predecessor candidate the start point of the current segment v_i is taken into consideration and for successor candidates the end point. The goal is to find a set of disjoint paths that maximizes the sum of the weights of the edges in the solution. The solution contains the corresponding vertices of these paths as an ordered set of vertices for each path which can then be merged to one continuous polyline representing a path.

A more refined variation of a segment graph will be presented in Section 5.2.

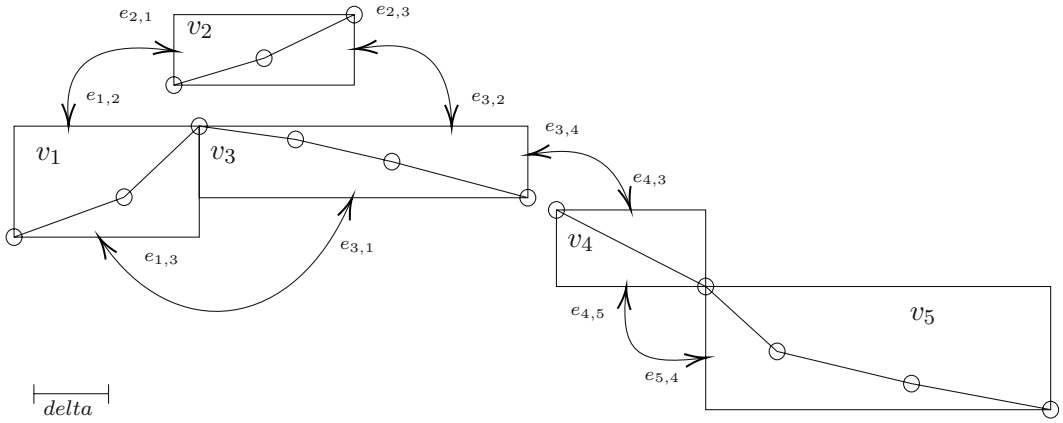


Figure 4.2: Data structure of the current algorithm

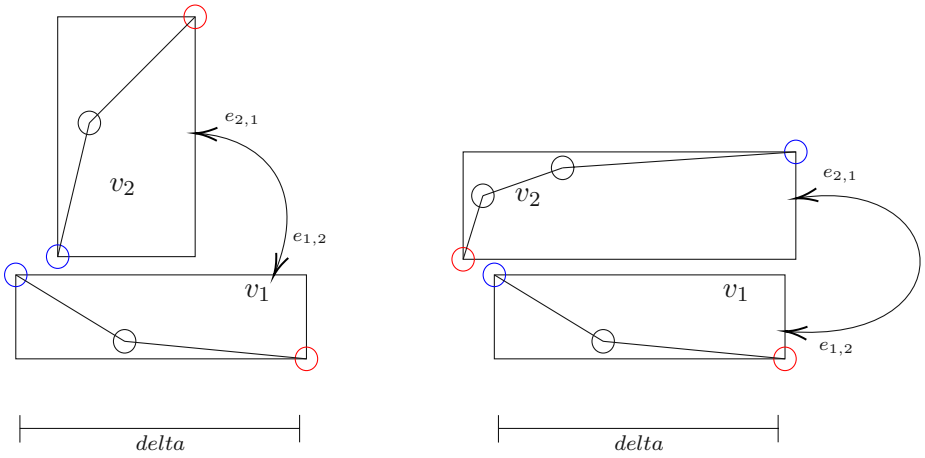
4.2 Weight Function

The weights ω of the edges of the segment graph are calculated by the distance weight ω^{dist} multiplied with the angle weight ω^{angle} multiplied by the intersection weight ω^{int}. The distance weight ω^{dist} is simply 1 minus the distance between the two segments. The angle weight ω^{angle} is $1 - |\frac{\alpha}{\pi}|$ where $\alpha_{i,j}$ is the deflection angle between v_i and v_j. The deflection angle is the angle between two straight lines formed by the first and second points or by the last and penultimate points of v_i and v_j respectively. And the intersection weight ω^{int} is 1 if there is an intersection between the two segments and 0.9 if not. The weight function should be improved.

$$\omega_{i,j} = \omega_{i,j}^{dist} * \omega_{i,j}^{angle} * \omega_{i,j}^{int}$$

4.3 Possible Problems while Creating the Segment Graph

Usually the current algorithm for generating the segment graph works well. However, when segments are just slightly larger than δ or self intersecting the current approach might lead to faulty heuristics. While creating the segment graph with its weighted edges, the algorithm first checks on a given Vertex v which successor candidates exist and adds them if all the conditions are met. Example 1 in Figure 4.3a presents a possible scenario where v_1 is given and the algorithm checks for successor vertices to append. For illustration purposes the start point of the segments are displayed in blue color and the end points in red color. Vertex v_2 will be added as a successor since it is in the neighborhood of v_1 (under the assumption that the weight of this edge will be higher than the defined threshold). The weight of the edge is determined by the weight function and since the candidate v_2 is first checked as a possible successor it will be added into the segment graph as such. The distance weight of the weight function will be calculated by using the end point of v_1 and the start point of v_2 instead of the start point of v_1 and the start point of v_2, which would result in a much higher weight and is the more realistic



(a) Example 1 (b) Example 2

Figure 4.3: Segment graph where the "wrong" coordinate from the bounding box is used for weight calculation

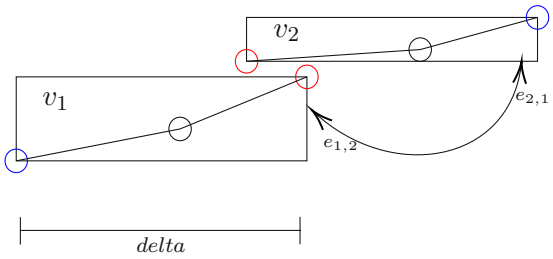


Figure 4.4: Segment graph where the "wrong" coordinate from the bounding box is used for the weight calculation

scenario. Currently, the algorithm would add both edges $e_{1,2}$ and $e_{2,1}$ once v_2 is selected as a successor candidate meaning that the other possibility is not even considered and the wrong weights are applied.

As another example, take a segment graph which is depicted in Example 2 in Figure 4.3b. Here the algorithm takes v_2 as the successor of v_1 because the start point of v_2 is in the neighborhood of the end point of v_1. As a result wrong weights would be stored.

In Figure 4.4 the algorithm selects v_2 as a successor candidate of v_1. However, in this specific case the function which determines $\omega_{1,2}$ and $\omega_{2,1}$ takes the start point which is further away and less realistic in the real world. Generally for a candidate v_j of v_i the algorithm always checks first if the start point of the candidate is in the neighborhood of v_i and uses this point for the weight function. Only if the start point is not in the neighborhood then the end point of v_j is checked respectively.

Similar to Figure 4.4, the wrong point of v_2 is taken for the calculation of $\omega_{1,2}$ and $\omega_{2,1}$ in Figure 4.5. Here v_2 is a predecessor candidate since it is not in the neighborhood of

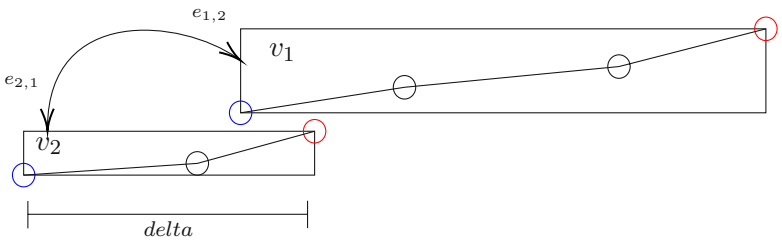


Figure 4.5: Segment graph where the "wrong" coordinate from the bounding box is used for the weight calculation

the end point of v_1.

A simple solution to slightly improve cases in which another segment is both a successor and a predecessor candidate: First just check the weights of the edges which are about to be inserted. If the adjacent vertex is only a successor candidate or only a predecessor candidate, insert the edge to the graph. If the adjacent vertex is both successor and predecessor candidate insert the edge with the higher weight. Currently, the successor candidate is always taken first if possible.

4.4 Selecting the Best Paths

After filtering out segments that are smaller than δ and creating the segment graph, the greedy track creation algorithm takes a random vertex of the graph which is not visited yet, marks it as visited and tries to prepend segments. It then always selects the segment with the highest probability, i.e. the edge with the largest weight, and marks it as visited. This is done until no further segments can be prepended. After that, the algorithm tries to append as many segments as possible while always choosing the one with the highest probability. Once no segments can be appended the current path, containing all the segments which were pre- and appended, is added to the solution set. An example for a path resolution can be seen in Figure 4.6. Unvisited segments are illustrated in blue, visited segments in red. In this example we assume that the path is expanded starting with v_3, which is marked as visited. We also assume that the start point of each segment is on the left while the endpoint is on the right side. The algorithm will first add segment v_2 to the path followed by v_1 marking them as visited. At this point no further segment with high enough probability (weight value greater than the threshold ω_{min}) can be appended to the path. The algorithm then proceeds to append the next vertex since no further segments can be prepended to the path. Segment v_4 is then appended, marking it as visited as well. After adding v_4 to the path, no further segments can be pre- or appended. The next unvisited vertex is then selected and the process of pre- and appending other segments is repeated until no unvisited vertices are left. A threshold ω_{min} can be set to only allow segments with a higher probability than ω_{min} to be pre- or appended. By default, the value of ω_{min} is set to 1/2. A pseudocode of this algorithm can be seen in Algorithm 4.2.

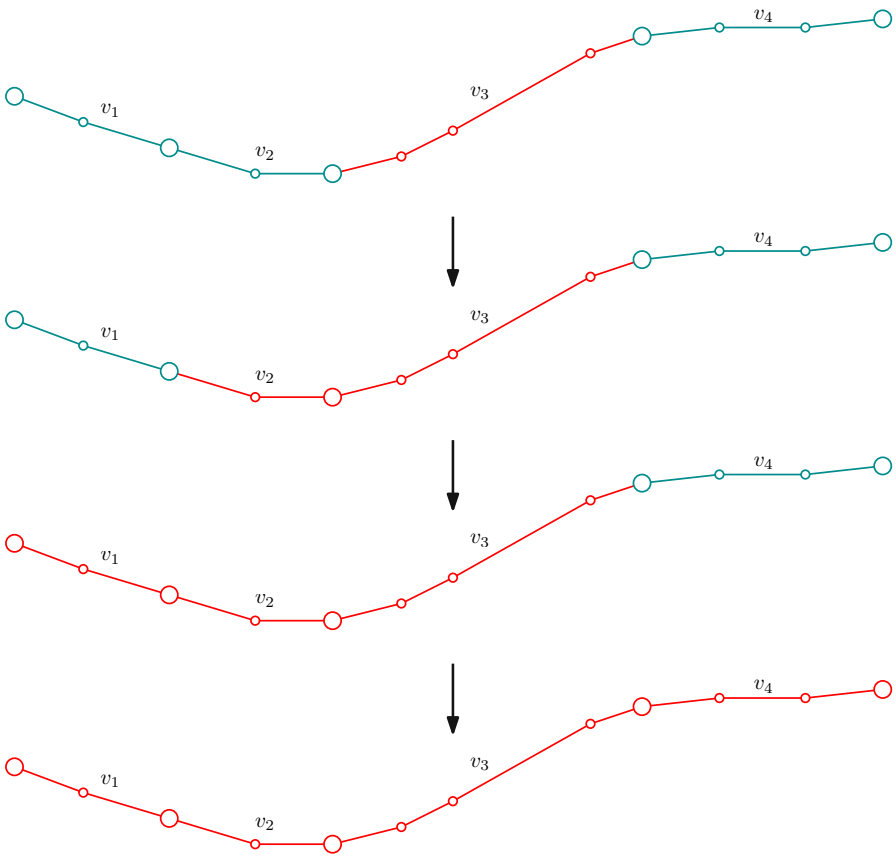


Figure 4.6: Example of a single path resolution with the track creation algorithm

The complexity of the simple algorithm is $\mathcal{O}(|E|)$ or $\mathcal{O}(|V|^2)$ respectively.

4.5 Possible Problems Selecting the Best Path

The current implementation has no backtracking. It merely searches for the next segment with the highest probability. In the case of Figure 4.7, assuming that the vertices are stored in numeric order (with v_1 being first and v_6 being last and v_1 being the first selected vertex), a path from v_1 to v_3 will be expanded since v_3 is more likely to be appended to v_2 than v_4. The expanded path is marked in red in Figure 4.7. After the expansion, vertices v_1, v_2 and v_3 are already visited, and the algorithm finds no other vertex which could be appended to v_3, since v_4 is not in the neighborhood of the end point of v_3. The algorithm then continues with expanding a new path v_4-v_5-v_6. The solution set would be $\{\{v_1, v_2, v_3\}, \{v_4, v_5, v_6\}\}$. A possibly better solution would be to expand the path v_1-v_2-v_4-v_5-v_6 and returning $\{\{v_1, v_2, v_4, v_5, v_6\}, \{v_3\}\}$ which cannot be found with the current approach. The situation in Figure 4.7 could for example happen if v_3 was an editing error and mistakenly added.

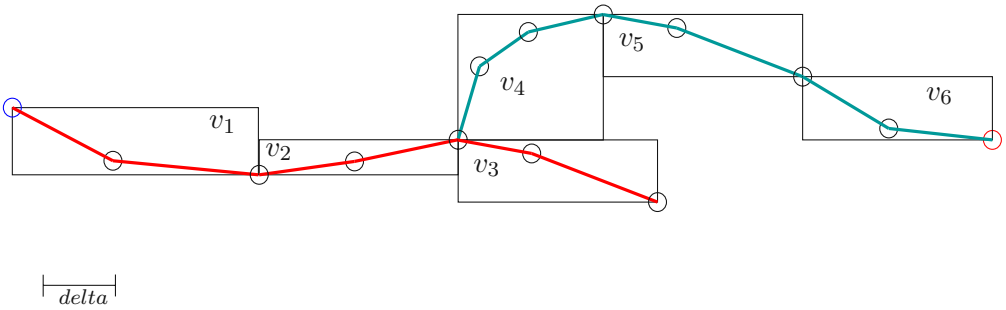


Figure 4.7: Expansion of a suboptimal path

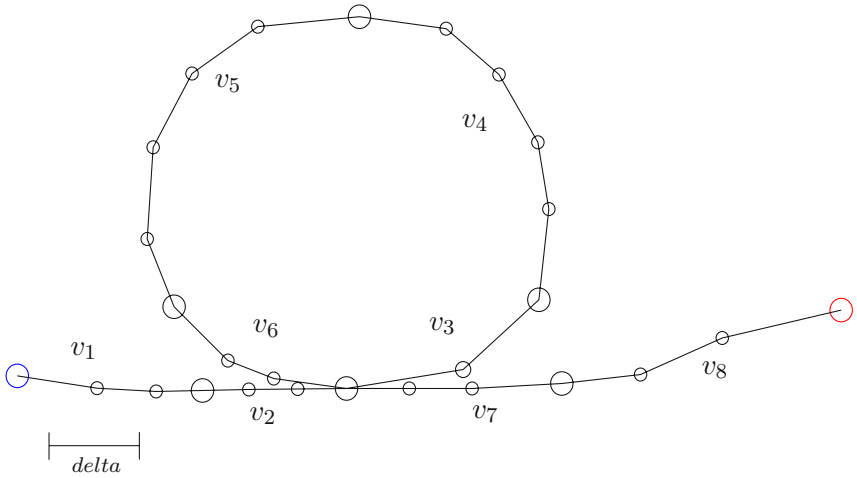


Figure 4.8: Expansion of the wrong path in a self intersecting track

Another situation where the current implementation meets its limits is one in which the track is self intersecting and contains a loop. Take Figure 4.8 as an example. Here we assume that all the segments are connected, and we omit the bounding boxes of segments for illustration purposes. Additionally, we mark the end and start points of segments with larger circles. Assuming that the vertices are numerically ordered again, the algorithm first searches for all the possible segments it can append to v_1. Segment v_2 is chosen as the first successor, but since the weight of $\omega_{2,7}$ is higher than $\omega_{2,3}$ the algorithm will prefer v_7 as the next successor. Thus, it will expand v_1-v_2-v_7-v_8 first and then continues with expanding v_3-v_4-v_5-v_6 as a separate path. The solution the algorithm returns is $\{\{v_1, v_2, v_7, v_8\}, \{v_3, v_4, v_5, v_6\}\}$.

4.6 Post-processing of the Paths

After the track creation, some segments in a path can be facing in the wrong directions. The reason for that is, that the original order of points within a segment is preserved when a segment is appended or prepended to a path, rather than the segment being reversed.

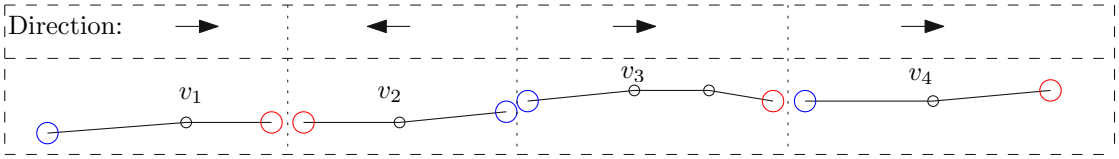


Figure 4.9: Example of inconsistent segment directions

A reference to the segment end point of the previously added segment is kept and in the next iteration the algorithm searches for the neighbors of the opposing end point of the newly added segment. Figure 4.9 shows inconsistent directions of segments. After applying the track creation algorithm each resulting path is only one single polyline. The order or sequence of points is crucial to produce the correct line, hence some adjustments of directions need to be applied for each segment before merging to a single polyline. To counteract that, a common direction is applied to all segments after the track creation algorithm is completed. In the proposed solution the direction of each segment is checked by comparing each segments start and end point to the first position of the whole track. Let S be the first point in a path. For each segment in a path, the distance between the start point of the segment and S is compared to the distance between the end point of the segment and S. If the end point is closer to S, then the segment will be reversed.

This approach of applying a common direction will lead to problems when a segment is part of a loop or strong (more than 90°) curve. Figure 4.10 shows how the directions would be applied to the segments if the track creation algorithm had returned the best path for the track in Figure 4.8. Segments v_4 and v_5 have faulty directions since the start points of those segments are further away from the start point of the path then their end points are.

We suggest improving this approach by comparing the distance of the start and end point of the segment to be added to the end point of path that has been discovered so far. For this, the new end point of the current computed path needs to be calculated and stored after each step. Since this is just a single additional operation it will barely affect performance while avoiding the problem shown in Figure 4.10.

After applying a common direction, line smoothing and simplification is applied. Line smoothing and simplification removes sharp edges and distortions in the paths that the track creation algorithm produces. It creates a smoother line, which will make the trajectory of the track more visually appealing and look more realistic when rendered. A good overview for line generalization including smoothing and simplification is given my McMaster in [18] and in the line generalization module of University of Alcalá [6].

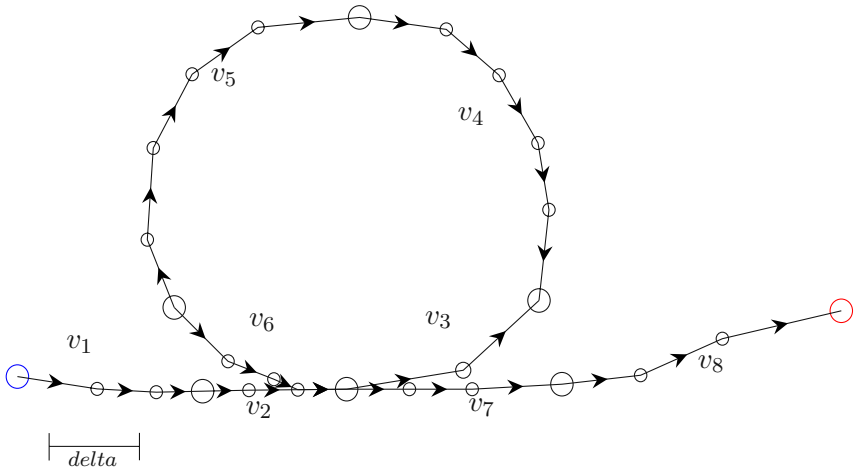


Figure 4.10: Applying common direction on self intersecting paths may cause problems

Algorithm 4.2: Current implementation of the track creation algorithm

Input : *segments*: A set of polylines with their metadata
Data: *segmentGraph*: $G_{seg} = (V, E)$
Result: All vertices from a spanning forest over the *segmentGraph*, segmented by its spanning trees (of components from the segment graph), where the edges of every spanning tree within the spanning forest strictly form a *path* ;
;

1 $segments \leftarrow \{s \mid s \in segments \wedge s \geq \delta\}$;
; /* Filter segments smaller than δ */
2 $G_{seg} \leftarrow createsegmentGraph(segments)$;
3 $visited \leftarrow \{\}$;
4 $result \leftarrow \{\}$;
5 **while** $v_{root} \leftarrow pop(\{v_i \mid v_i \in V \wedge v_i \notin visited\})$ **do**
6 $v_f \leftarrow v_{root}$;
7 $continousSegmentString \leftarrow \{\}$; /* Dequeue */
8 **while** $v_f \neq \emptyset$ **do**
9 $candidates \leftarrow \{v_i \mid v_i \notin visited \wedge (\omega_{i,f} > \omega_{min})\}$;
; /* Find candidates to prepend */
10 $next \leftarrow \{v_i \mid v_i \notin visited \wedge (\omega_{i,f} = max(\{\omega_{j,f} \mid v_j \notin visited\})) \wedge v_i \in candidates\}$; /* Select the predecessor candidate with the highest probability */
11 $continousSegmentString.addFirst(next)$;
12 $visited \leftarrow visited \cup next$;
13 $v_f \leftarrow next$;
14 **end**
15 $v_b \leftarrow v_{root}$;
16 **while** $v_b \neq \emptyset$ **do**
17 $candidates \leftarrow \{v_i \mid v_i \notin visited \wedge (\omega_{b,i} > \omega_{min})\}$;
; /* Find candidates to append */
18 $next \leftarrow \{v_i \mid v_i \notin visited \wedge (\omega_{b,i} = max(\{\omega_{b,j} \mid v_j \notin visited\})) \wedge v_i \in candidates\}$; /* Select the successor candidate with the highest probability */
19 $continousSegmentString.addLast(next)$;
20 $visited \leftarrow visited \cup next$;
21 $v_b \leftarrow next$;
22 **end**
23 $result \leftarrow result \cup continousSegmentString$;
24 **end**
25 **return** *result* ;

CHAPTER 5

Graph extraction from OSM

In OpenStreetMaps there are three different types of objects: *nodes*, *ways* and *relations*. *Tags* are maps (key-value pairs) that store additional information about objects. Nodes typically represent point locations in 2d-space which are defined by their coordinates. They represent elements such as points of interest (*POIs*), amenities, or they can be part of a way or relation. Ways are ordered lists of nodes which typically represent linear features such as street segments, rivers, or railway track segments. Relations are ordered lists of nodes, ways and other relations which typically represent more complex features such as a public transport route or a boundary. A whole railway track is represented by a relation which contains all the segments that belong to the track. In theory, if the data is tagged correctly, it should be pretty easy to convert the data into other commonly used formats like GeoJSON. However, more often than not, the data is not tagged correctly, contains errors or is inconsistent. Listing 5.1 shows an example of a way element from the OSM in XML format. Each OSM element contains an `id`, an 64-bit integer which is unique among all elements from the same element type. The `changeset` describes on which database operation the element has been added or modified, `timestamp` defines the time at which this operation happened and `uid` the id of the user that created or last modified this element. Sub elements within ways named `nd` hold a reference to a node element. Usually a way element holds multiple `nd` elements that define the geometry of the way element. Tags are defined as sub elements within a way. Additional information they hold are for example the maximum speed `maxspeed`, the operator name `operator` or the railway type `railway`, as shown in Listing 5.1. Some properties of the way shown in Listing 5.1 are omitted in this example for brevity. Listing 5.2 shows an example of a node element which is part of the way from Listing 5.1. Most importantly, node elements hold position information in `lat` and `long` that are also essential for the geometry of the way element that references the node.

```
<osm version="0.6"
   copyright="OpenStreetMap and contributors"
   attribution="http://www.openstreetmap.org/copyright"
      ⋮
>
<way id="130556301"
   version="13"
   changeset="140751908"
   timestamp="2023-09-03T10:35:43Z"
   uid="4453208"
      ⋮
>
   <nd ref="148975337"/>
   <nd ref="11164817209"/>
   <nd ref="11164817208"/>
   <nd ref="1172206522"/>
   <nd ref="1172206153"/>
   <tag k="electrified" v="contact_line"/>
   <tag k="maxspeed" v="60"/>
   <tag k="operator" v="OEBB-Infrastruktur AG"/>
   <tag k="railway" v="rail"/>
   <tag k="voltage" v="15000"/>
            ⋮
</way>
</osm>
```

Listing 5.1: Example of a OSM way

5.1 Fetching Railway Segments from the Raw Data

Data directly from OSM can either be fetched as XML or as a Protocol Buffers (Protobuf) format. Both XML and Protobuf are data formats that are commonly used to serialize and exchange data, but other formats such as GeoJSON might be more suitable for further processing of the geometric data. Overpass [21] is a database that is built on top of OSM and provides an API that can be used to fetch OSM data with a query language like syntax. Additionally, the queried data from Overpass can be converted into GeoJSON. GeoJSON is a file format to represent geographical features with JSON syntax. The geometry objects in GeoJSON are called features and consist of a geometry and additional properties. A multitude of features can be combined into a feature collection.

One of the most popular API's for querying OSM data is the Overpass API [21]. For

```
<osm version="0.6"
   copyright="OpenStreetMap and contributors"
   attribution="http://www.openstreetmap.org/copyright"
      ⋮
>
<node id="11164817209"
   visible="true"
   version="1"
   changeset="140751908"
   timestamp="2023-09-03T10:35:43Z"
   uid="4453208"
   lat="48.2873231"
   lon="14.2893915"
      ⋮
/>
</osm>
```

Listing 5.2: Example of a OSM node

the querying from Overpass, one has to create a query that corresponds to the syntax of the Overpass Query Language (OverpassQL). From the total set of data in OSM, we can query three different types of objects: nodes (representing points in the map and can be part of ways), ways (linear features representing street segments, railway track segments, rivers etc.) and relations (complex features such as public transport routes). In Overpass syntax these are specified by the keywords of the same name i.e. `node`, `way` and `relation` in the query. By providing filters in the query, the result set can be narrowed down to the desired data. A filter in a query is indicated by square brackets and contains a key-value pair that specifies the tag and which values should be filtered. As an example `[railway=rail]` filters all ways that have the tag `railway` with the value `rail`. Overpass also supports querying certain areas, which can be specified with the `area` keyword. This queries just data from Austria:

```
area["ISO3166-1"="AT"];
```

Alternatively, one can also query regions by providing a bounding box within a filter with the `bbox` keyword. The expression `[bbox=47.1,11.4,47.2,11.5]` queries data from the bounding box with the coordinates 47.1, 11.4, 47.2, 11.5. The output format from the query can also be specified with the `out` keyword. Adding `out body`, for example, returns all data with additional information like tags and coordinates, whereas `out skel` only returns the id and coordinates of the data. Since we create a purely geometry-based algorithm, we need the result of `out skel` most of the time. Another

statement we use is `>;` which basically returns the objects that are below the current object in the hierarchy. So for a way it returns all nodes that are part of the way and for a relation it returns all ways and nodes that are part of the relation. In our case, if we want to query all nodes that are ways that have the tag `railway` with the value `rail` in an area, let it be "47.1, 11.4, 47.2, 11.5", we can use the following query:

```
way(area:47.1,11.4,47.2,11.5)[railway=rail];
out body;
>;
out skel;
```

As specified in [24], railway segments should be tagged with the key `railway`. And the value `rail` should be reserved for "regular railway tracks". These values for querying railway segments can be modified to fetch tracks and the query can be broadened or narrowed down according to the needs.

In the beginning of the query we can also define the output format (the format in which the data will be represented in the result). Defining `[out:json];` at the beginning of the query returns the data in JSON format.

After the querying from Overpass, we want to further process the data in Java. Consequently, we parse the GeoJSON objects from the Overpass query and convert them to JTS geometries. The JTS Topology Suite [16] is a Java library that provides data structures and algorithms for processing geometry objects. Furthermore, it supports representation of objects in other coordinate systems like *UTM* (Universal Transverse Mercator). Therefore, using JTS as an object model for the geometries can be of advantage if we want to use data from other providers that do not use the *WGS 84* (World Geodetic System 1984) coordinate system. All GeoJSON objects are defined in the WGS 84 coordinate system [3]. WGS 84 is a geodectic coordinate system that uses latitude and longitude to determine positions of points on the earth's surface and has degrees as units. Projected coordinate systems such as UTM, on the other hand, typically use meters as distance units and might therefore be a more intuitive system to present data in. The specific unit has to be taken into account when we use distance parameters such as δ. A translation between the two coordinate systems is also possible as outlined in [17]. OSM Ways are represented as JTS LineStrings and Nodes as JTS Points in the algorithm's data structures. Once completed with the processing of the data (applying our clustering algorithm), we convert the JTS geometries back to JSON/GeoJSON objects.

5.2 Converting the Data to a Segment Graph

In order to hold the segments and their possible connections we need to create an appropriate data structure. A graph containing all the segments seems to be a promising structure. There is a multitude of other projects that are using graphs to represent OSM data. A selection of them can be found in 2.1. For this we construct a segment graph

structure that we briefly introduced in Section 4.1. In order to keep the complexity of the instances of the segment graph as low as possible, we will reduce the number of vertices and edges we add to this graph as much as possible without harming the structure's integrity too much. Additional preprocessing steps remove all segments that are unlikely to be part of a feasible solution and merge segments that would unambiguously be connected to each other in any feasible solution. The merges and removals of segments should lead to a smaller instance and boost the performance of our algorithm significantly.

5.2.1 STR-Tree

In this subsection we will briefly discuss a data structure we use for spatially indexing the 2d-geometries in our algorithm, the STR-tree. The STR-tree uses the Sort-Tile-Recursive algorithm for packing an R-tree (see Section 3.1).

Since we already have a predefined set of geometry objects (the segments), the input instances of our algorithm, which are fairly static and do not change much during the further steps of our algorithm, it is advantageous to preprocess the index records of the R-tree by organizing and sorting them in a way to construct an R-tree which is more efficient in terms of loading, space utilization and query performance. A simple and effective algorithm for this preprocessing or packing is proposed by Leutenegger et al. [14], the Sort-Tile-Recursive algorithm. We use the STR-tree implementation provided by the JTS Topology Suite [16] for indexing our geometries. It is used both for the merging of unambiguously connected segments and for the actual generation of the segment graph.

5.3 Merging of Unambiguous Segments

One of the biggest contributing factors to the runtime of the path finding algorithm is the complexity of the input instance. In general, the more segments there are, the bigger the complexity. The complexity is accelerated by the amount of segments that are neighboring each other. If we translate the input instance into a segment graph described in Section 4.1, we can see that the complexity of each node in the corresponding segment graph is proportional to the number of neighboring segments. We achieve a significantly smaller input instance by merging unambiguous segments in a preprocessing step. Two segments are unambiguous if they only have each other as single predecessor or successor candidate respectively. We define a predecessor candidate as a segment that is in the neighborhood of the starting point of the segment and a successor candidate as a segment that is in the neighborhood of the end point of the segment. Pseudocode for the merging of unambiguous segments is listed in Algorithm 5.1. Firstly, all segments from the segment graph are marked as *mergeable*. As shown in line 4 of Algorithm 5.1 onwards, for every segment v_s that has only one predecessor candidate v_s^{pre} in δ range of the start point of v_s, we check if the connected endpoint of the predecessor candidate $v_s^{pre}.conn$ also only has exactly one neighboring endpoint that is v_s. If so, we merge v_s and v_s^{pre}, otherwise we mark v_s not *mergeable* to avoid looping over the same segment again. In the merging step we create a new segment v_s^{mer} and remove the old segments

Algorithm 5.1: Merge Segments Algorithm

Input : Segment graph $G_{seg}= (V, E)$
Result: Updated segment graph G_{seg} after merging

1 **foreach** $v_s \in V$ **do**
2 $\quad mergeable(v_s) \leftarrow true$;
3 **end**
4 **while** $\exists v_s(v_s \in V \wedge (|predecessors(v_s)| = 1) \wedge mergeable(v_s))$ **do**
5 $\quad$ **if** $|neighbors(v_s^{pre}{}_{.conn})| = 1 \wedge v_s \in neighbors(v_s^{pre}{}_{.conn})$ **then**
6 $\quad\quad v_s^{mer} \leftarrow merge(v_s, v_s^{pre})$;
7 $\quad\quad V \leftarrow \{V \cup v_s^{mer}\} \setminus \{v_s \cup v_s^{pre}\}$;
8 $\quad\quad E \leftarrow E \setminus \{e_{s,s^{pre}} \cup e_{s^{pre}s,}\}$;
9 $\quad\quad G_{seg} \leftarrow (V, E)$;
10 $\quad\quad G_{seg} \leftarrow updateEdges(G_{seg})$
11 $\quad$ **end**
12 $\quad$ **else**
13 $\quad\quad mergeable(v_s) \leftarrow false$
14 $\quad$ **end**
15 **end**

v_s and v_s^{pre} and their edges from the segment graph. On line 10 we recalculate the edges for v_s^{mer} and neighboring segments. The next iteration of the while loop will already contain the new segment v_s^{mer}. At the end all segments and edges in the segment graph will be updated with the merged segments and new edges.

The pre-merging of segments has the big benefit of reducing the complexity of the input instance for many cases. However, in particular for one case that frequently occurs on railway tracks, the simple segment pre-merging does not provide a satisfiable result. The case that two tracks are running parallel for a long distance and the endpoints of the individual segments of both tracks are at approximately the same position. When this happens, no merging can be performed, since there is always more than one neighbor for each segment. And the complexity of the algorithm will rise drastically. To counteract that we introduce *merging of parallel running segments* in Algorithm 5.2. We start by marking all segments as *mergeable*. For each segment v_s that has exactly three predecessor candidates v_s^{n1}, v_s^{n2}, v_s^{n3} in δ range of the start point of v_s, we check if the neighbors v_s^{n1}, v_s^{n2}, v_s^{n3} have exactly the other two segments and the original point as neighbors from their respective connecting point. We select the neighbor with the highest probability v_s^{max} (line 12) and merge it with v_s in line 18. The other two neighbors are then also merged with each other in line 19. The new segments are added to the vertices of the segment graph and the old segments are removed. Then the edges of the new segments are recalculated in line 22. If one of the neighbors does not exactly have the other three segments as neighbors from their respective connecting point or the weight between the two neighbors is below ω_{min} the segment v_s is marked as not *mergeable* in order to avoid

Algorithm 5.2: Merge Parallel Segments Algorithm

Input : Segment graph $G_{seg}= (V, E)$
Result: Updated segment graph G_{seg} after merging

foreach $v_s \in V$ **do**
  $mergeable(v_s) \leftarrow true$;
end
while $\exists v_s(v_s \in V \wedge (|predecessors(v_s)| = 3) \wedge mergeable(v_s))$ **do**
  if $\exists v_i(v_i \in V \wedge v_i \in \{neighbors(v_s^{n1}{}_{.conn}) \cup neighbors(v_s^{n2}{}_{.conn}) \cup neighbors(v_s^{n3}{}_{.conn})\} \wedge v_i \neq v_s \wedge v_i \neq v_s^{n1} \wedge v_i \neq v_s^{n2} \wedge v_i \neq v_s^{n3})$;
  then
    $mergeable(v_s) \leftarrow false$;
    continue at 4 ;
  end
  $v_s^{max} \leftarrow max\{\omega_{s,s_{n1}}, \omega_{s,s_{n2}}, \omega_{s,s_{n3}}\}$;
  let v_s^{o1} and v_s^{o2} be the other two segments (in arbitrary order);
  if $\omega_{s_{o_1},s_{o_2},<} \omega_{min}$ **then**
    $mergeable(v_s) \leftarrow false$;
    continue at 4 ;
  end
  $v_s^{mer_1} \leftarrow merge(v_s, v_s^{max})$;
  $v_s^{mer_2} \leftarrow merge(v_s^{o1}, v_s^{o2})$;
  $V \leftarrow \{V \cup v_s^{mer_1} \cup v_s^{mer_2}\} \setminus \{v_s \cup v_s^{n1} \cup v_s^{n2} \cup v_s^{n3}\}$;
  $G_{seg} \leftarrow (V, E)$;
  $G_{seg} \leftarrow updateEdges(G_{seg})$
end

looping over the same segment again and the algorithm continues with the next segment.

After the merging of unambiguous segments we prune all segments that are not connected to any other segment.

To illustrate a possible merging process, let us have a look back at Figure 4.8. It is pretty obvious that v_1 and v_2 have to be connected. We can further observe that v_3, v_4, v_5 and v_6 have to be connected, too, and v_7 and v_8 respectively, as there is only one feasible solution to join them. All of these segments only have a maximum of one predecessor candidate which is in the neighborhood of its starting point and a maximum of one successor candidate which is in the neighborhood of the end point. We will merge all the neighboring segments with this property, under the condition that the weight between each two successive segments is sufficient. This avoids creating redundant vertices and edges in the segment graph and the algorithm will explicitly tackle the cases in which more than one possibility of joining two segments exist.

A positive side effect of doing a merging of segments before creating the graph is, that in

some cases the set of possible solutions can be reduced. An example where this happens is if we apply the just mentioned merging on the input of Figure 4.8. The resulting segments are (v_1, v_2), (v_3, v_4, v_5) and (v_6, v_7, v_8). Since (v_3, v_4, v_5) is now just one segment, we can eliminate the possibility of this merged segment to be connected to itself. This not only reduces the possibilities of how all the segments can be connected, but also leads to a better result in cases of single loops like the one shown in Figure 4.8.

5.4 Constructing Solutions from the Ordered Segments

After merging and ordering the track segments, we need a way to construct presentable solutions from them. QGis [22] is a project that provides visualization capabilities for GIS data. The project supports the importing of GeoJSON files, which can be used to visualize the solutions if we convert the results of the algorithm's solution into GeoJSON format. Additionally, QGis provides different styling options for GeoJSON features that we want to take advantage of. In case further processing of the data is desired, an option is to convert the result of the algorithm (a set of sets of ordered track segments) back to a graph structure. This structure can be arbitrary and is not required to have the same format as the segment graph used for the algorithm. One could for instance create a graph where the vertices represent the end points of segments and the edges represent the control points of the respective segments. Such a graph allows for further powerful calculations on the data.

CHAPTER 6

Experimental Algorithm

6.1 Local Continuity Weight Function

In Section 4.2 we briefly described a simple approach on a weight function for defining the likeliness of two segments to be concatenated. Every edge in the segment graph is associated with a weight which is determined by the weight function. Additionally, the exact (end) points that connect the two segments are stored as well. First we will choose the two connecting end points which have the highest probability. There are a maximum of four possibilities on how to connect two segments. Section 4.3 shows different examples with more than one possibility to connect two segments. We will be satisfied with the possibility which scores the highest weight. All possibilities will be checked. The old weight function had a distance weight, angle weight and an intersection weight and was calculated by: $\omega_{i,j} = \omega_{i,j}^{dist} * \omega_{i,j}^{angle} * \omega_{i,j}^{int}$. We remove the intersection weight because whether an intersection between two segments is present or not, is not a huge indicator if two segments are connected. Instead of the angle weight we introduce a new metric which we call *continuity*. The continuity is calculated using a spline approximation of the segments and comparing the slopes at the segment ends. In the weight function defined in Section 4.2 the distance weight is calculated by: $\omega_{i,j}^{dist} = 1 - distance(i,j)$. This distance weight is adjusted to be more realistic and to avoid negative probabilities. It is scaled by the min weight and delta value. We also add a third metric ω^{rel} that we call *relative distance weight*. In this metric we assign weight 0 if the distance between the two segments is larger than half the length of segment itself. Otherwise, we set the value to it to 1. Similarly to the weight function presented in Section 4.2, the new weight function is: $\omega^{dist} * \omega^{cont} * \omega^{rel}$. In the following V_s is the segment from which the connection originates and V_t is the segment to which the connection is made.

6.1.1 Distance Weight

Previously, the distance was calculated by 1 minus the distance of the two end points in meters. By this definition the distance weight can even go below 0 which, in that case and assuming that the angle and intersection weights are positive, will return a negative value as the total edge weight. To avoid subsequent errors and make it more realistic (there should not be negative probabilities, just 0 probability instead), we will normalize the distance weight. We also might want to make the allowed distance variable and not constrict it to 1 meter. Here we might even use the delta value and normalize the weight over delta. The distance weight is defined as:

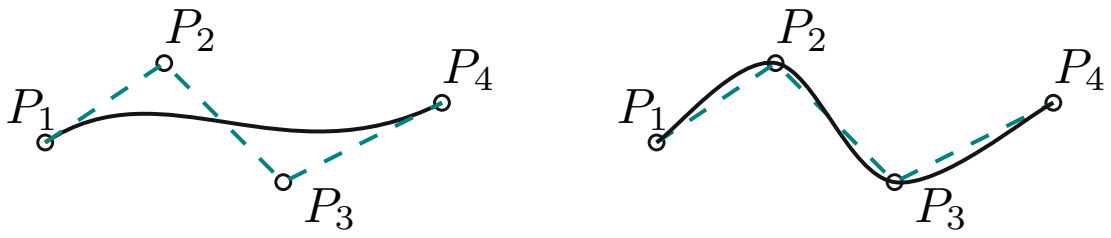
$$\omega^{dist} = max\left\{0, \left(1 - \frac{distance * \omega_{min}}{\delta}\right)\right\} \tag{6.1}$$

Most notably, the distance weight is now scaled by the min weight and delta value. This allows the weight to be distributed more evenly if the actual distance is below delta. And it makes all distances smaller than delta to have a higher weight than the min weight.

6.1.2 Continuity approximation with splines

Besides the distance, the relative angle of two segments to each other was one major factor in the weight function from Section 4.2. To calculate the angle at a possible connection point of two segments, the weight function from Section 4.2 constructed lines using the last and second to last point of the segments at the connecting endpoints. The angle weight, which is a value between 0 and 1, was defined as: $1 - |\frac{angle}{\pi}|$. In a lot of cases just using the angle at the intersection for the weight function is not sufficient. Figure 4.8 shows such an example. It therefore can be helpful to consider the whole polyline of a segment and its trajectory in order to approximate the likeliness of it being connected to another segment. For creating curves over the points in the polyline we can use splines. Determining the continuity of this generated spline might be a better measure than the pure angle at the intersection point. We implement this idea in for the improved weight function.

The track segments are stored as an ordered set of points. Using these points to create a curve might approximate the real shape of the track segment better than just a polyline. We can create a spline as a mathematical representation of this curve using the points of the segment as the control points for the spline. There are many variations of splines like bézier spline, hermite spline or cardinal spline. (A linear spline would be equivalent to the current representation of segments, a polyline.) For our problem we want the spline to pass through all the control points, so we need an interpolating spline. The cardinal spline seems to fit our purposes well. In particular the catmull-rom spline, which is type of cardinal spline where the tension value is 0.5. One problem we face is that the catmull-rom spline uses the previous and the following point as a measure to determine the slope at a given control point. This is exactly for the first and the last control point problematic. At the first control point the previous point needs to be estimated and the following point at the last control point respectively. Figure 6.2 shows an example



(a) B-spline (Bézier Spline) (b) Cardinal Spline

Figure 6.1: B-spline and cardinal spline

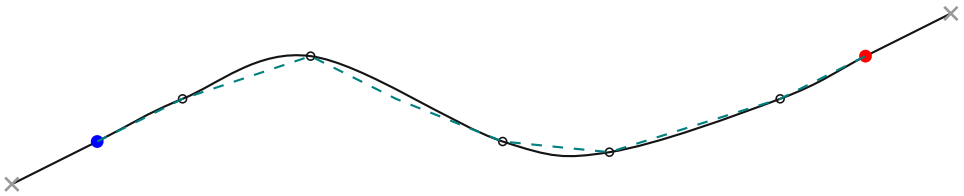


Figure 6.2: Example of spline with interpolated start- and end point

of a spline where the start and end point are interpolated. The start- and end point are colored in red and blue respectively, the interpolated points are depicted as a gray cross. There are different types of parameterization at the control points. The centripetal parameterization might be suitable since it avoids cusps and self-intersections on curve segments between control points according to [29].

On these splines we can calculate the derivative and then check if the resulting curve is continuous (parametric continuity). If the original curve is continuous it is called $\mathcal{C}^0$ continuous. If it is $\mathcal{C}^0$ continuous and continuous on the first derivative it is $\mathcal{C}^1$ continuous, if it is $\mathcal{C}^1$ continuous and continuous on the second derivative it is $\mathcal{C}^2$ continuous and so on. We can also calculate the tangents at given locations of the spline, for example at control points, to check if the curvature of the spline is continuous (geometric G^1 continuity). By comparing the tangents and derivatives of the two end points of neighboring segments, we can measure the continuity of the two segments. Checking the continuity of the constructed splines might provide a good value for the weight function.

In our approach to calculating the continuity between two segments, we leverage the catmull-rom spline, a type of interpolating spline characterized by its ability to pass through all control points. This property, combined with the spline's capacity to effectively represent a path, makes it an ideal choice for our purposes. In order to calculate the continuity of a spline at a given point we require at least four control points per definition. The interpolated value can then be calculated within the inner two control points. Since we want to calculate the value at the end point of a segment, we need to add a control point to the end of the segment, which is an estimation of the next point. To calculate this point we simply calculate the difference of the coordinates from the last point to the second last point and add it to the last point. For instance given three control points

P_0, P_1 and P_2, we would calculate the fourth (estimated) control point P_3 as follows: $P_3 = P_2 + (P_2 - P_1)$. The continuity is calculated by the angle of the tangents at the end points of the two interpolated splines from the two segments. The angle or slope is calculated by the derivative of the spline at the end point. Following equation defines the position of a catmull-rom spline at a given point t:

$$q(t) = 0.5 * ((2 * P_1)+ \tag{6.2}$$
$$(-P_0 + P_2) * t+ \tag{6.3}$$
$$(2 * P_0 - 5 * P_1 + 4 * P_2 - P_3) * t^2+ \tag{6.4}$$
$$(-P_0 + 3 * P_1 - 3 * P_2 + P_3) * t^3) \tag{6.5}$$

Where $P_0 - P_3$ are the control points of the spline and t is the value at which we calculate the interpolated point. Parameter t is 0 at the start point of the spline and 1 at the end point of the spline. We calculate the first derivative of this equation to get the tangent at a given point t:

$$q'(t) = 0.5 * ((P_2 - P_0)+ \tag{6.6}$$
$$(2 * P_0 - 5 * P_1 + 4 * P_2 - P_3) * 2 * t+ \tag{6.7}$$
$$(-P_0 + 3 * P_1 - 3 * P_2 + P_3) * 3 * t^2) \tag{6.8}$$

The continuity is then calculated by the angle between the two tangents at the end points of the two splines. For calculating the continuity between the end of a segment v_1 and start of v_2, let $q'_1(t)$ be the tangent at the end point of the spline created from v_1 and $q'_2(t)$ be the tangent at the start point of the spline created from v_2, then the continuity weight is calculated by:

$$\omega^{cont} = 1 - \left| \frac{angle(q'_1(1), q'_2(0))}{\pi} \right| \tag{6.9}$$

6.1.3 Relative Distance

If a segment is connected to another small segment and the distance between those two segments is greater than half of the length of the small segment, then the small segment is unlikely to be part of the real track geometry and likely an editing error or does not add significant information to the geometry of the track. We want to omit the small segment then. This is done by following function which returns the relative distance weight:

$$distance^{rel} = \begin{cases} 0 & \text{if } distance(V_f, V_t) > \frac{length(V_t)}{2} \\ 1 & \text{otherwise} \end{cases} \tag{6.10}$$

Algorithm 6.1: Exhaustive Line Expansion

Input : Segment graph $G_{seg}= (V, E)$
Result: A set of paths P (with maximal weight) that cover the segment graph G_{seg}.

```
1  P ← {};
2  U ← V;
3  while U ≠ ∅ do
4  |   possibleSolutions ← {};
5  |   foreach v_r ∈ U do
6  |   |   possibleSolutions =
   |   |    possibleSolutions ∪ (exhaustiveLineSearch(G_seg, v_r, U, {}));
7  |   end
8  |   bestSolution ← max {ω(possibleSolutions)};
9  |   P ← P ∪ bestSolution;
10 |   U ← U \ bestSolution;
11 end
12 return P ;
```

6.2 Segmenting the Graph into Long Paths

In the following approach we iterate over all plausible concatenations of track segments to find the "best" solution. The solution should be a set of vertex-disjoint paths of maximum total weights and cover the segment graph.

We will discuss core parts of Algorithm 6.1 in simplified form.

At the beginning we define a set of unvisited vertices U which contains all vertices of the segment graph G_{seg}. Further, we introduce a function *exhaustive line search* described in Algorithm 6.2, that finds the eligible path with maximum weight starting from a vertex v_i. We call the *exhaustive line search* for every unvisited vertex v_i and store the path with maximum weight along with the segments (vertices) that are part of the path and the total weight in a set of possible solutions. With all the possible solutions the algorithm then selects the solution with the maximum weight, adds it to the result set and removes all vertices that are part of the solution from the set of unvisited vertices U. This process is repeated until U is empty.

The *exhaustive line search* function from Algorithm 6.2 starts with at a given vertex v_r and searches for the path with maximum weight in the segment graph G_{seg}. Vertices already marked as visited are omitted from this search. At first, we introduced a recursive algorithm that performs a depth first search in both directions. The recursion of the algorithm breaks if there are no more unvisited neighbors for v_r. In that case, the algorithm returns just v_r with a weight of 0 on line 17. In any other case for every neighbor that is a potential candidate v_c of vertex v_r, the currently selected vertex v_c will be appended or prepended to the current path and the weight between v_c and v_r ($\omega_{c,r}$ or

Algorithm 6.2: Exhaustive Line Search (Recursive)

Input : Segment graph: $G_{seg}= (V, E)$
v_r: The vertex from which the search starts.
visited: A set of vertices are already part of a solution.
P: The path that has been expanded so far with $\omega(P)$ being the accumulated weight of the path so far.

Result: An ordered set of segments and its combined weight.

```
1  visited ← visited ∪ v_r;
2  poss ← {} // Set of possible solutions
3  all_candidates ← predecessor_candidates(v_r) ∪ successor_candidates(v_r);
4  foreach v_c ∈ all_candidates do
5      if v_c ∈ predecessor_candidates then
6          P.addFirst(v_c);
7          ω(P) ← ω(P) + ω_{c,r};
8          poss ← poss ∪ exhaustiveLineSearch(G_seg, v_c, visited, P);
9      end
10     else
11         P.addLast(v_c);
12         ω(P) ← ω(P) + ω_{r,c};
13         poss ← poss ∪ exhaustiveLineSearch(G_seg, v_c, visited, P);
14     end
15 end
16 if poss = ∅ then
17     return P;
18 end
19 else
20     return {solution | solution ∈ poss ∧ ω(solution) = max {ω(poss)}};
21 end
```

$\omega_{r,c}$) is added to the total weight of the solution. Predecessor candidates $predecessor(v_r)$ are unvisited adjacent segments within δ range to the start point of v_r with $\omega_{c,r} >= \omega_{min}$ and successor candidates $successor(v_r)$ are unvisited adjacent segments within δ range to the end point of v_r with $\omega_{r,c} >= \omega_{min}$. The algorithm then calls itself recursively on line 8 (or 13) with v_c as the new root vertex. The current part of the solution (path with total weight) is also passed to the recursive call along with the updated set of visited vertices. Vertex v_c from the caller is then also added to the list of visited vertices. After all the possible solutions (with its recursive calls) are added to a list, the solution with the maximum weight of this list is returned in line 20.

This recursive approach delivered the correct solution. However, due to the amount of recursive calls it became quite slow with increasing complexity of the segment graph.

We then tried to improve the algorithm by using an iterative approach in Algorithm 6.3.

In this approach we use a stack (line 2) for storing the current solutions and its assigned weights, and loop over this stack until it is empty. We store the best solution which has weight 0 at the beginning and only v_r as a vertex as seen in line 3. We also keep track of the segment v_l that was newly added to the latest solution in the stack and its connected endpoint. In every iteration we pop the last element (solution) from the stack and check if it is a better solution than the current best solution. If it is, we update the best solution in line 7. For each neighbor v_c of the latest added vertex that is not visited or contained in the current solution we add a new current solution containing the previous solution appended or prepended with v_c and the total weight of the previous solution plus the weight between v_c and the previous vertex to the stack of current solutions. Since we kept track of the endpoint that the latest added vertex in the solution we popped, we can now determine if we want to append successor candidates or prepend predecessor candidates to the current solution. If the endpoint of the last added segment was the start point, we only search for successor candidates, otherwise we only search for predecessor candidates. A neighbor is a predecessor candidate $predecessor(v_l)$ if it is an unvisited adjacent segment within δ range to the start point of v_l with $\omega_{c,r} >= \omega_{min}$ and a successor candidates $successor(v_l)$ is an unvisited adjacent segment within δ range to the end point of v_r with $\omega_{r,c} >= \omega_{min}$. If the latest added vertex v_l has no eligible neighbor, no new solution is added to the stack. After the loop terminates the best solution is returned. This iterative approach results in a slight performance increase.

The path resolution in this approach is only one directional. A new segment will only be appended or prepended to the last added segment. To ensure that all possible paths are explored, the exhaustive line search algorithm is called for every vertex in the segment graph that is not already part of another solution.

Algorithm 6.3: Exhaustive Line Search (Iterative)

Input : Segment graph: $G_{seg}= (V, E)$
v_r: The vertex from which the search starts.
visited: A set of vertices are already part of a solution.

Result: Best solution *best_solution* which contains an ordered set of segments and the total weight.

```
1  v_l ← v_r // last added segment
2  path_stack ← {v_r};
3  best_solution ← {v_r};
4  while path_stack ≠ ∅ do
5  |  path ← path_stack.pop();
6  |  if ω(path) > ω(best_solution) then
7  |  |  best_solution ← path;
8  |  end
9  |  visited ← visited ∪ {v | v ∈ path};
10 |  all_candidates ← predecessor_candidates(v_l) ∪ succesor(v_l);
11 |  foreach v_c ∈ all_candidates do
12 |  |  if v_c ∈ predecessor_candidates(v_l) then
13 |  |  |  path_stack.push(path.addFirst(v_c));
14 |  |  |  ω(path) ← ω(path) + ω_{c,l};
15 |  |  end
16 |  |  else
17 |  |  |  path_stack.push(path.addLast(v_c));
18 |  |  |  ω(path) ← ω(path) + ω_{l,c};
19 |  |  end
20 |  end
21 end
22 return best_solution;
```

CHAPTER 7

Experiments

In this chapter we will present and discuss the results of the experiments we performed with the algorithms and their evaluation. By setting up an experimental setup with a variety of data, edge cases and different parameters we try to cover a wide range of possible scenarios.

7.1 Data

The primary requirement for our experiments is the generation, collection and preparation of diverse input instances. To achieve this, we will produce input instances from different regions, and data providers such as OSM. In addition to the instances fetched from real world data providers, we will also generate smaller input instances that cover edge cases.

We tested the algorithms on these tracks:

- **Astana Karaganda:** The Astana Karaganda dataset contains all track segments in an area around Astana and Karaganda in Kazakhstan.
- **Haramain Track:** This track is part of the Haramain High Speed Railway in Saudi Arabia. It connects Mekka with Medina. Some track segments do not have an assigned track id making an import by track id not possible.
- **Alishan Track:** The Alishan track is a section of the Alishan Forest Railway Network, a narrow gauge railway network in Taiwan which covers mountainous areas and forest.
- **OEBB tracks:** The OEBB dataset [10] contains all the main tracks of the Austrian railway network. The dataset is obtained directly from the main railway operator OEBB of Austria. This dataset is used in the case study we present in Section 7.6.

- **Areas:**
  - **Bischofshofen:** A dataset containing all track segments from an area in Austria between Bischofshofen and Altenmarkt. The track segments are fetched from OSM with the bounding box `[47.273839, 13.145142, 47.445045, 13.474731]`.
  - **Barcelona:** Track segments from OSM in the area of Barcelona with the bounding box `[41.347309, 2.067490, 41.433208, 2.287216]`.
  - **Texas:** Track segments from an area between Austin and Dallas fetched from OSM with the bounding box `[29.522325, -98.617782, 29.648899, -98.402374]`.
- **Custom generated tracks:**
  - **Self intersecting track scenario:** This scenario represents an edge case where the corresponding track intersects itself. The intersection point is located at the endpoints of multiple track segments. This scenario is modeled after the example explained in Figure 4.8.
  - **Forked track scenario:** In this scenario the track splits into multiple tracks. Endpoints of track segments are at the location where the track splits. For switches on tracks this is a common scenario. Further this can be caused by an editing error in OSM (mistakenly adding a small segment/way to a track in OSM). We base this scenario on the example in Figure 4.7.
  - **Small overlapping segments scenario:** This scenario represents an inaccuracy in the location of the end and start points of track segments causing the segments have shifted endpoints and not sharing the same coordinates at the endpoints. This can be caused by measuring inaccuracies for the positions (GPS) or editing errors in OSM. We described these problems in the example in Figure 4.4 and Figure 4.5.
  - **Small segments scenario (1 & 2):** On instances with very small track segments or big delta values an issue might arise where the wrong endpoints accumulate a better score in the heuristic. The small segment scenarios represent these edge cases described in Figure 4.3.

7.1.1 Real word data

In the following table we list the amount of track segments and the average length of segments for datasets of real world tracks:

Track	No. Track Segments	Avg. Length of Segments
Astana Karaganda	40	7,869.59 m
Haramain Track	352	2,783.87 m
Alishan Track	97	780.36 m
Bischofshofen	384	360.55 m
Barcelona	632	384.02 m
Texas	1,078	1,650.61 m
OEBB Lines (Case Study)	1107	5,418.51 m

7.1.2 Generated data

In the following table we list the amount of track segments and the average length of segments for datasets of generated tracks:

Track	No. Track Segments	Avg. Length of Segments
Self intersecting track	8	548.38 m
Forked track	6	382.38 m
Small overlapping track	2	19.15 m
Small segments scenario1	2	9.93 m
Small segments scenario2	2	8.67 m

The generated tracks represent edge cases in which the segments lie in ambiguous geometric positions.

7.2 Experimental Setup

Each run with a specified custom area, defined by a bounding box, will be executed separately, whereas the runs where the input instance is provided by a GeoJson file will be executed automatically one after the other. To avoid race conditions or shared usage of resources the runs will be executed sequentially. To document the results of the experiments we implemented a benchmarking tool which records measures such as the runtime, size of the result, result quality etc. The results are written to a file.

By performing multiple experiments with the algorithms we will be able to evaluate and analyze them in more details. We want to measure:

- Longest path. (We assume that generally a longer solution is better. This however, might not always be the case. In some instances a longer solution might represent a more unrealistic track in terms of geometric properties.)
- Accuracy of the solution set on different input instances.
- Size of the solution set on different input instances.

- Performance on edge cases (like loops, ambiguous segments, ...).
- Runtime performance.
- Termination on big input instances.
- If a bigger solution set (longer individual tracks) is always a better solution.

Implementation and System Specifications:

Every experiment was run on Java 21.0.2 using JTS Version 1.18.2 and Spring Boot 3.2.2. The experiments were executed on a machine with an Intel Core i7-1260P CPU and 32GB of DDR4-3200 RAM. Data-fetching from OSM was done using Overpass API v0.7.61. The elapsed time for each run was measured in wallclock time.

7.3 Visualization of the Results

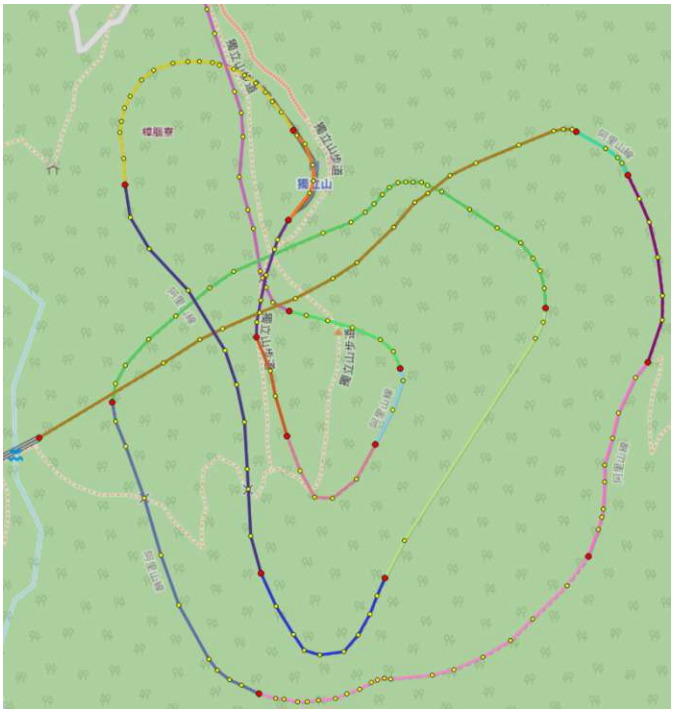
To visualize the output of the algorithms we will use the open source software QGIS. Track segments are imported as vector layers and then stylized with different properties. Segments are categorized such that each segment has a unique stroke color. For segments that indicate the input data we additionally mark the start and end point with a read marker and the control points with smaller yellow markers. For the output segments, arrows indicate the direction of the segments. An example of the visualization can be seen in Figure 7.1.

7.4 Comparing the Algorithms

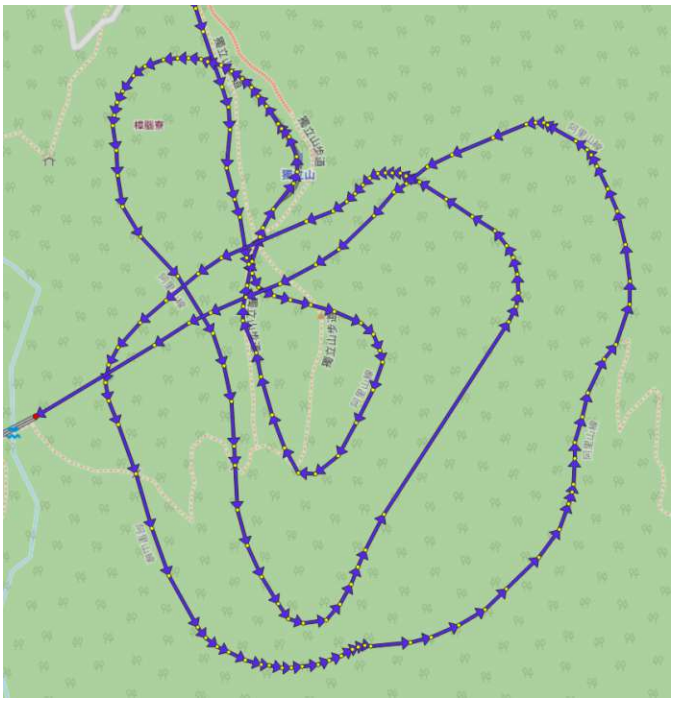
A central aspect of this work is to assess the introduced algorithm and evaluate potential improvements or weaknesses. We use the greedy local fit algorithm presented in Chapter 4 as a baseline and compare it to the different variations of the algorithm we propose in Chapters 5 and 6.

The different parameters that we vary in the experiments are:

- **min weight:** defines the minimum weight of a segment to be considered for the track creation
- **delta (δ):** defines the maximum distance between two segments to be considered for the track creation
- **segment pre-merging:** defines whether segments should be merged before the track creation
- **merge lookahead:** defines the lookahead for the merge operation
- **merge radius:** defines the radius for the merge operation



(a) Original track segments from the Alishan track section



(b) Algorithm output for the Alishan track section

Figure 7.1: Visualization (described in Section 7.3) of the original track segments and the algorithm output for the Alishan track section

Astana Karaganda Benchmark

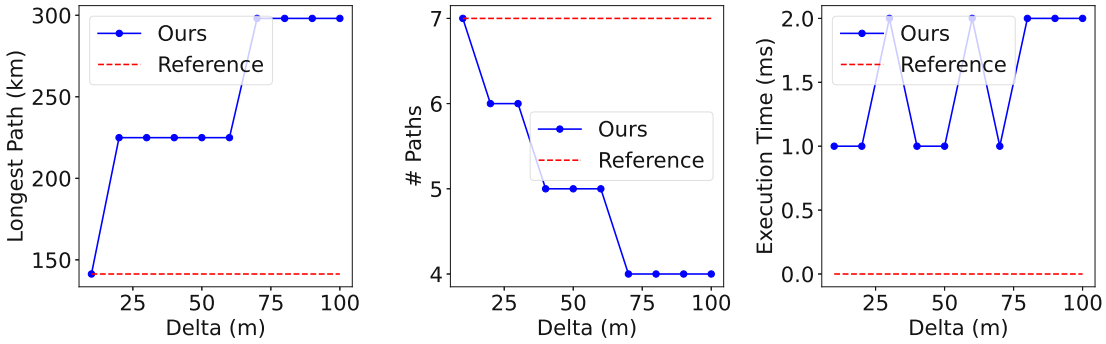


Figure 7.2: Run for Astana-Karaganda Track

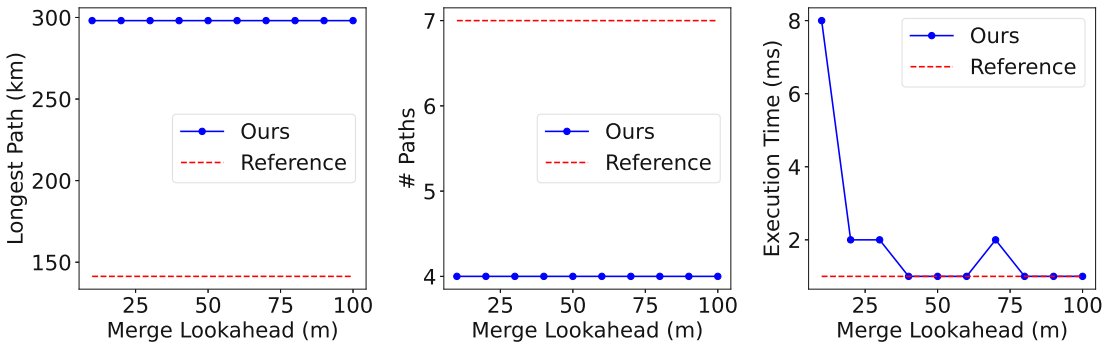


Figure 7.3: Run for Astana-Karaganda Track

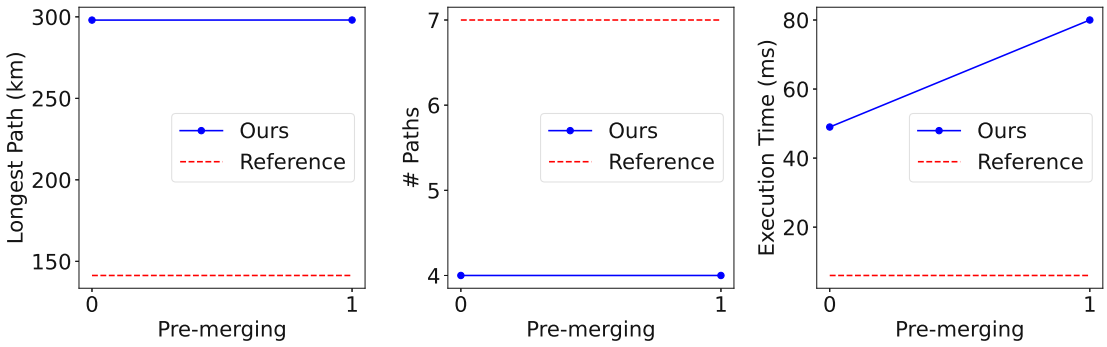


Figure 7.4: Run for Astana-Karaganda Track

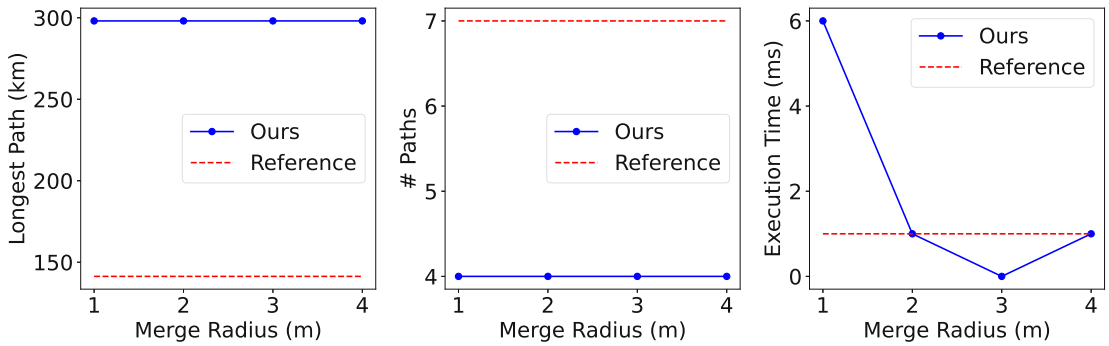


Figure 7.5: Run for Astana-Karaganda Track

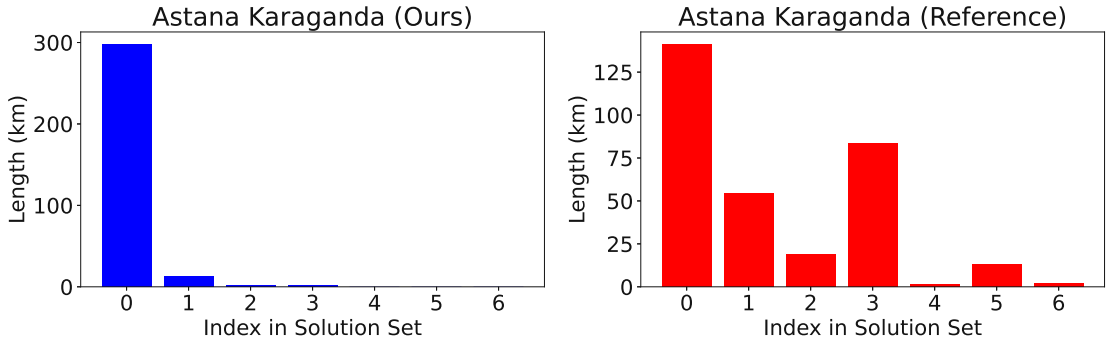


Figure 7.6: Length per Path

Haramain Benchmark

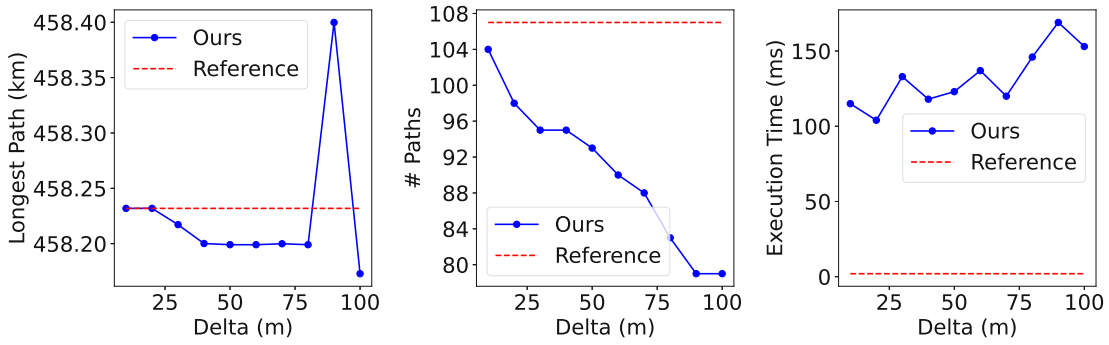


Figure 7.7: Run for Haramain Track

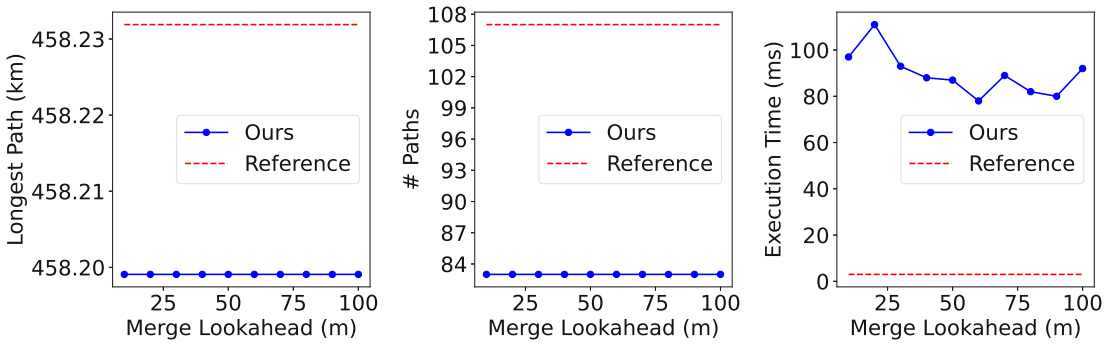


Figure 7.8: Run for Haramain Track

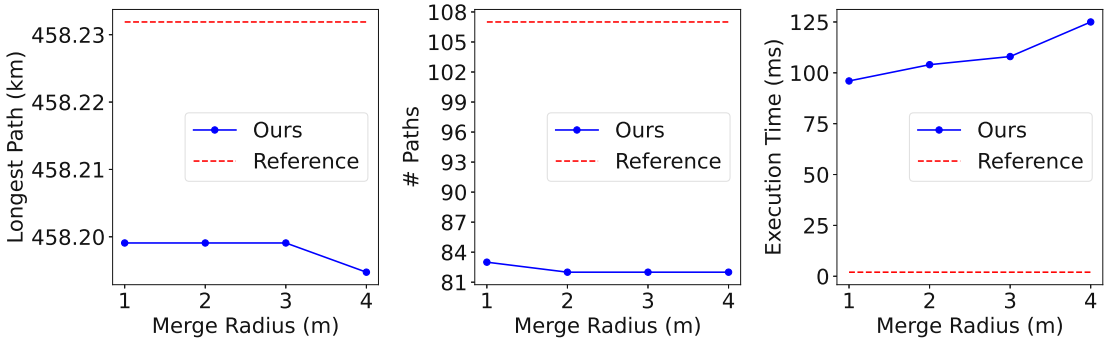


Figure 7.9: Run for Haramain Track

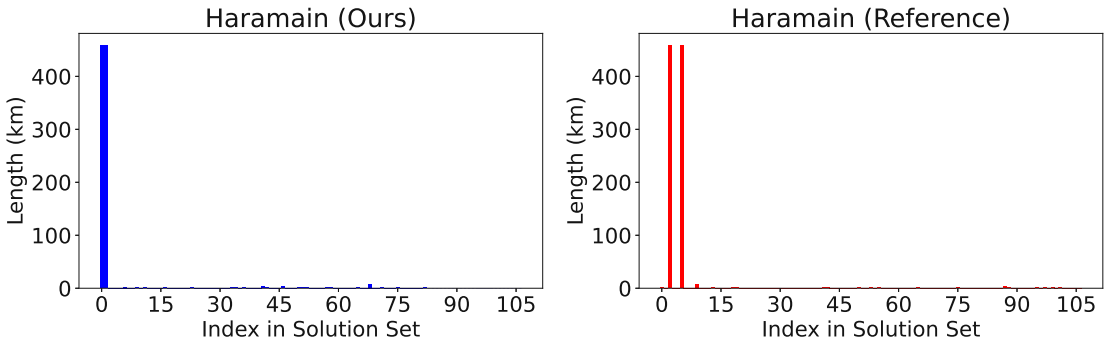


Figure 7.10: Length per Path

Alishan Benchmark

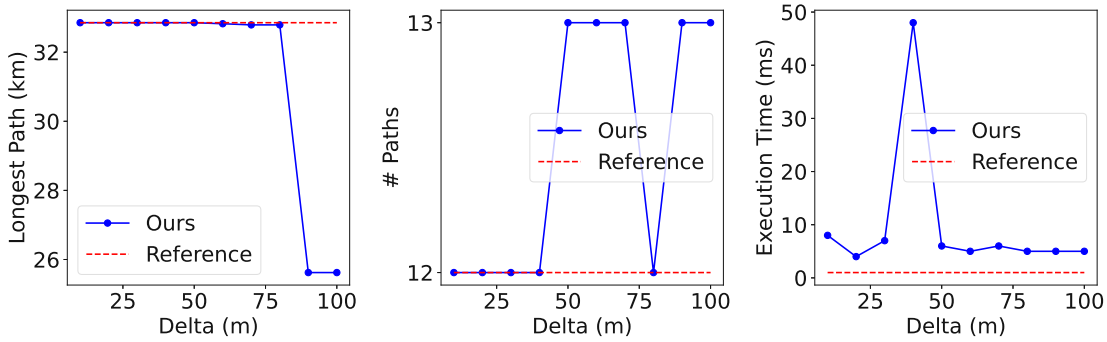


Figure 7.11: Run for Alishan Track

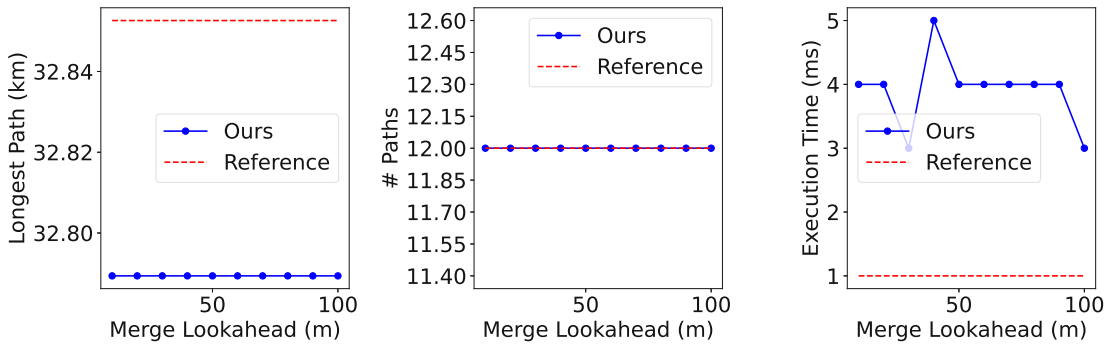


Figure 7.12: Run for Alishan Track

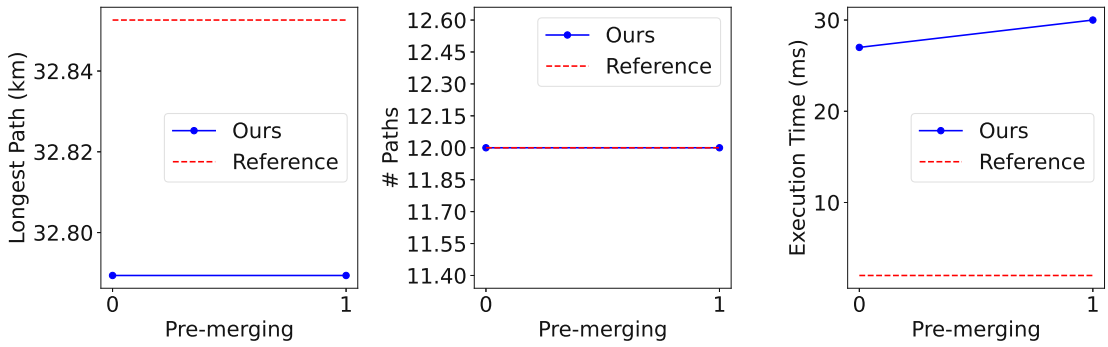


Figure 7.13: Run for Alishan Track

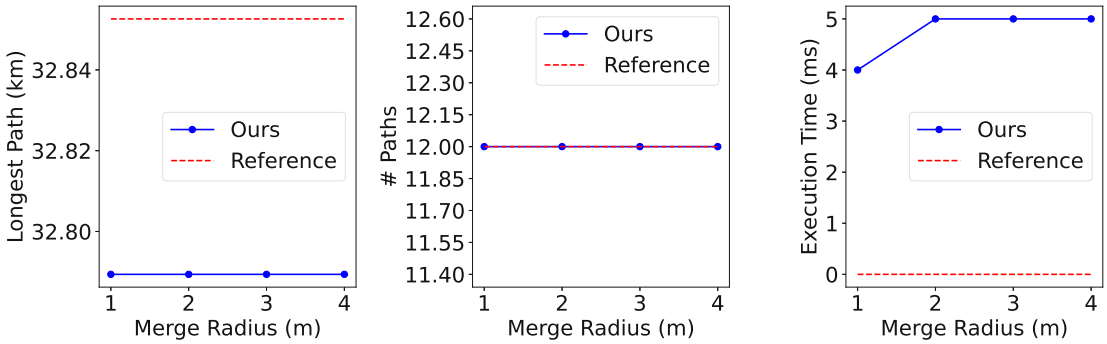


Figure 7.14: Run for Alishan Track

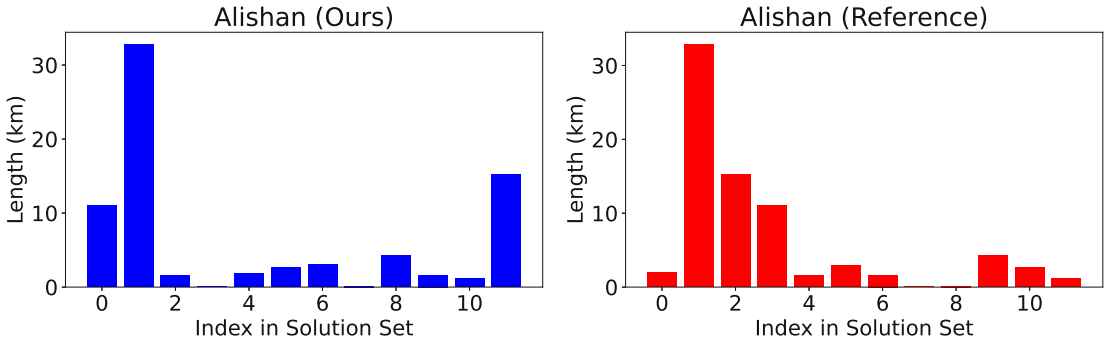


Figure 7.15: Length per Path

7.4.1 Discussion

The plots from Figure 7.2 through Figure 7.27 show the results of the experiments run on the datasets described in Section 7.1. Runs for each dataset with each variable parameter are sectioned into three different line plots and two bar plots. Within each line plot, the x-axis is the variable parameter that is observed in this run. The leftmost line plot depicts the longest found path (in km) on the y-axis. Concurrently, the center line plot shows the number of paths found (i.e. the size of the solution set). Lastly, the rightmost line plot shows the execution time of the run. We assume that the length of the longest path does not necessarily determine the quality or accuracy of the algorithm instance by itself. A longer path might easily be computed by merging segments which do not represent a single track realistically. Generally, however, computing a longer longest path is an indicator of a better solution. The cardinality of the solution set (number of unique paths found) that is depicted in the middle line plot of each run is a metric which we consider quite robust to measure the accuracy of our algorithm. A smaller cardinality typically depicts a more desirable outcome, as it minimizes the presence of unconnected or isolated segments. When coupled with the execution time of the algorithm, this metric offers a comprehensive assessment of algorithmic efficacy and quality. A shorter running time

and a lower number of paths in the solution set will typically signify a better strategy. Although minor disparities in execution time may be negligible, exponential increases in runtime for larger instances significantly reduced algorithmic effectiveness. In the bar plots we illustrate the length of each path that was found in the solution set. The x-axis of this plot shows the index of the corresponding path indicating the order in which the paths got discovered by the algorithm. The y-axis shows the length of each path in meters. We distinguish between the reference solution (red) and the new algorithm (blue).

Delta:

The delta value determines the radius for which the algorithm considers two individual segments as neighbors. For the step of segmenting the graph into long paths this is the most important parameter. We observe that with increasing delta value the number of paths in the solution set decreases or stagnates, with the Alishan track in Figure 7.11 being an exception. The longest path found also increases in distance in most cases, however, we can not safely draw a conclusion based on these values. As for the execution time, it stayed mostly the same with some deviations with increasing delta value. This could be due to small instances. We will further observe the execution time on larger instances in Section 7.5.

Pre-merging:

We observe that whether pre-merging is enabled or not, does not influence the solution significantly. Runs of the two smaller input instances, Astana Karaganda in Figure 7.4 and Alishan in Figure 7.13 have similar runtime both with and without pre-merging. For the Haramain track, which is a much larger input instance, we run into an issue where the algorithm does not terminate in adequate time. The algorithm with pre-merging enabled finished in a few seconds, while the run without pre-merging had to be terminated manually after approximately one hour. Following the merging step in the run with pre-merging enabled, the number of segments decreased from 352 in the original input instance to 91. Consequently, the segment graph's complexity, containing only 91 vertices, significantly shrunk, thus enabling a relatively fast execution.

Merge radius:

The merge radius parameter determines the maximum distance between two segments to be considered for the merge operation for pre-merging. We can see that a change in value of this parameter does not affect the solution set much. Only the runtime is also barely affected by the merge radius parameter on small instances.

Lookahead:

For merging parallel running segments the lookahead parameter determines the minimum distance for which two disjunct sets of segments have to be running in order to be

considered for the parallel merge operation. This parameter has also no effect of the solution set. Just the runtime varies slightly with the different lookahead parameter values.

Interestingly, we can see that for the Astana Karaganda input instance the result of the new algorithm seems to be completely different to the one from the reference solution as seen in Figure 7.15. The new algorithm finds a path with almost 300 km length, while the reference solution only finds a path with approximately 150 km. The Haramain track consists of two rails that run parallel to each other for a very long distance. This is reflected in the bar chart in Figure 7.10 that shows the length of each path in the solution set. The new algorithm and the reference solution both find the same two long paths that represent the two parallel rails. Both algorithms also perform close to the same on the Alishan track in Figure 7.15 where only the order of resolved paths slightly differs.

7.5 Running the Algorithms on Query Windows

To see how the algorithms perform when no track information can be derived from the data, we performed some runs on a set of all track segments in different areas.

The runs on query windows show similar results to the results of runs presented in Section 7.4. We can see that the runtime on larger instances drastically increases with growing delta value. In particular, from a delta value of around 80m onwards, the runtime seems to increase exponentially. The reference algorithm finds longer longest paths in the Barcelona and Texas area than the new algorithm. The new algorithm, however, finds a smaller set of solutions to cover all track segments and has on average more longer paths than the reference algorithm. Both merge radius and merge lookahead do not influence the performance of the algorithm much. Pre-merging disabled led to the algorithms not terminating in adequate time for the larger instances. The runs for Bischofshofen and Barcelona had to be canceled after around one hour each. The Austin-Dallas run in Figure 7.26 took more than 4 times longer with pre-merging disabled than with pre-merging enabled.

7.5.1 Bischofshofen - Altenmarkt

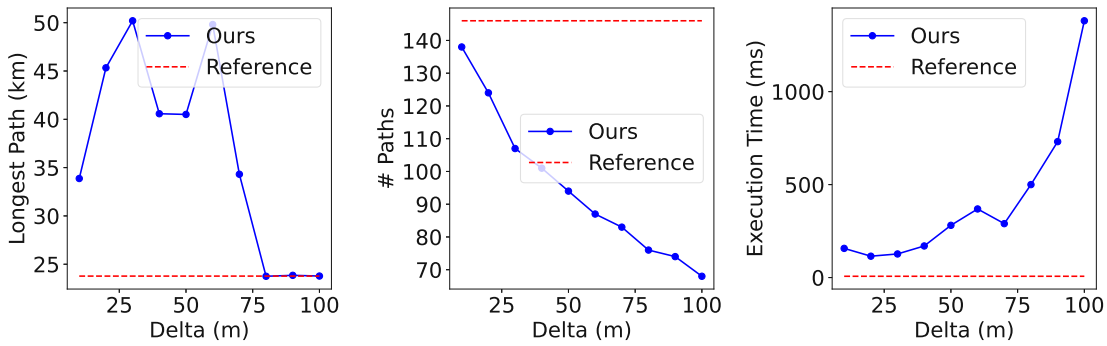


Figure 7.16: Run for Bischofshofen-Altenmarkt Area

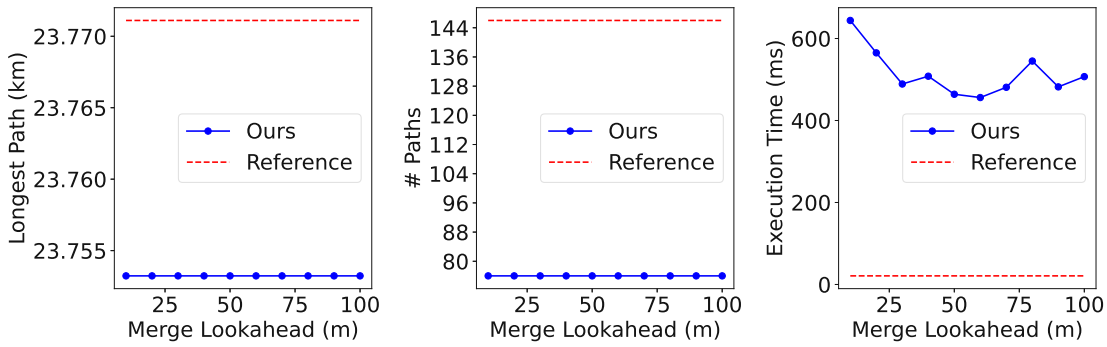


Figure 7.17: Run for Bischofshofen-Altenmarkt Area

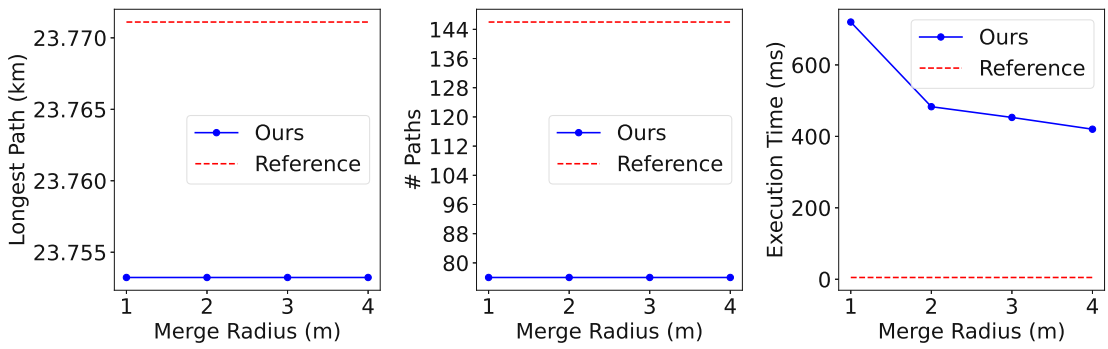


Figure 7.18: Run for Bischofshofen-Altenmarkt Area

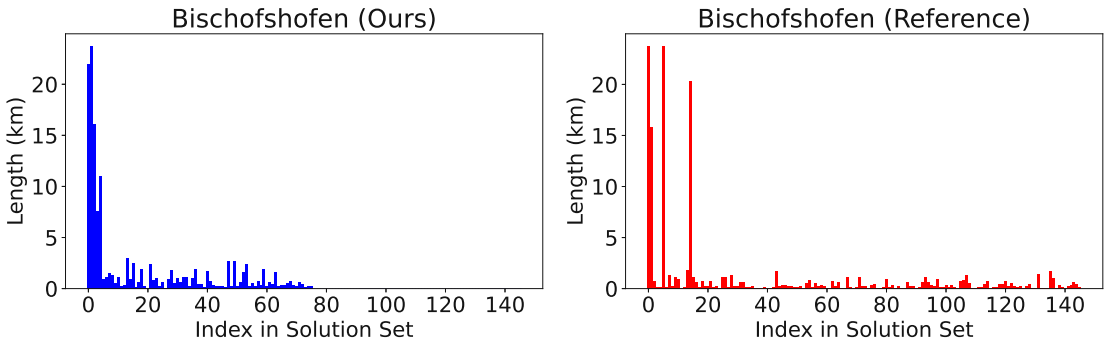


Figure 7.19: Length per Path

7.5.2 Barcelona

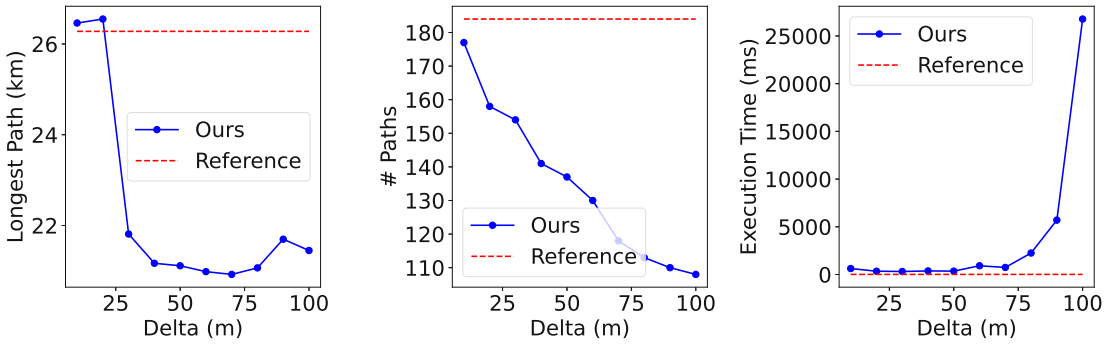


Figure 7.20: Run for Barcelona Area

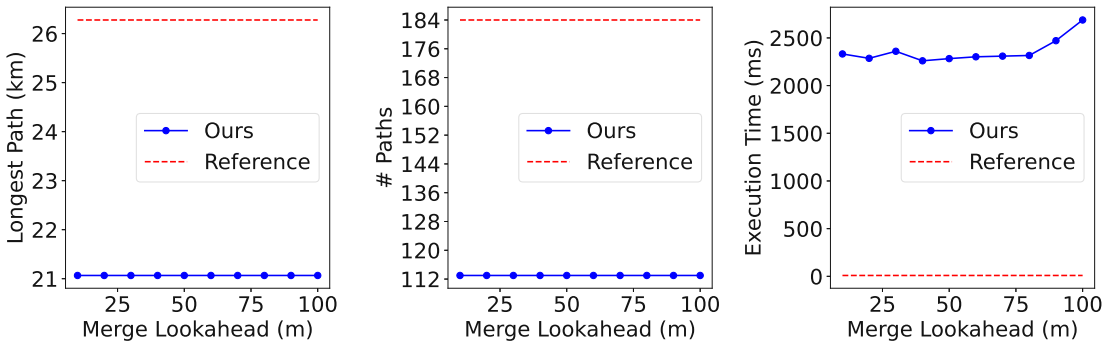


Figure 7.21: Run for Barcelona Area

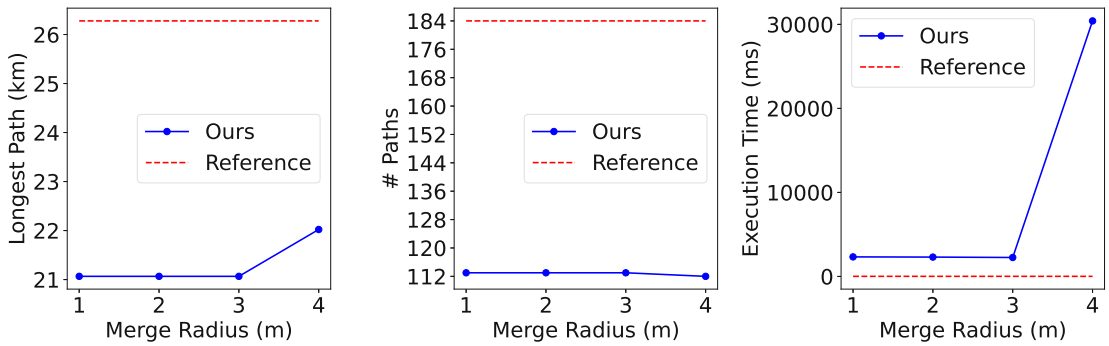


Figure 7.22: Run for Barcelona Area

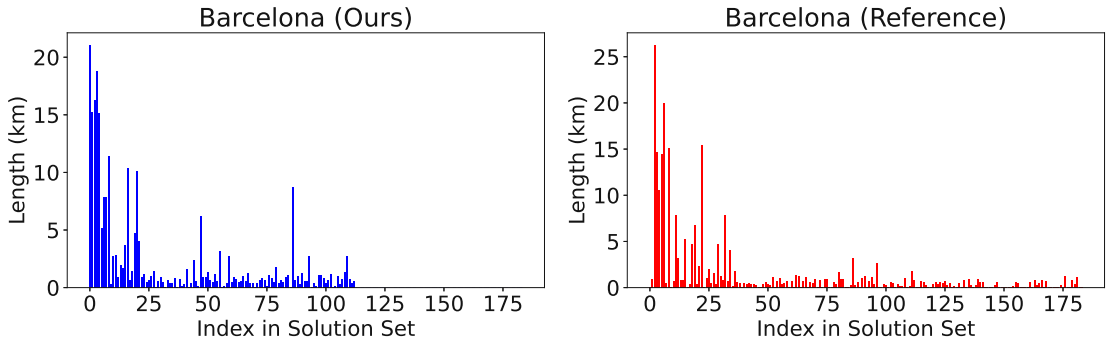


Figure 7.23: Length per Path

7.5.3 Austin-Dallas

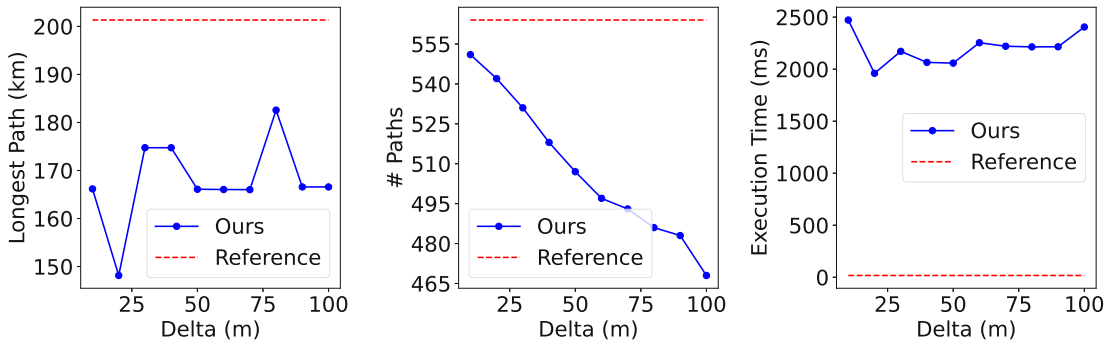


Figure 7.24: Run for Austin-Dallas (Texas) Area

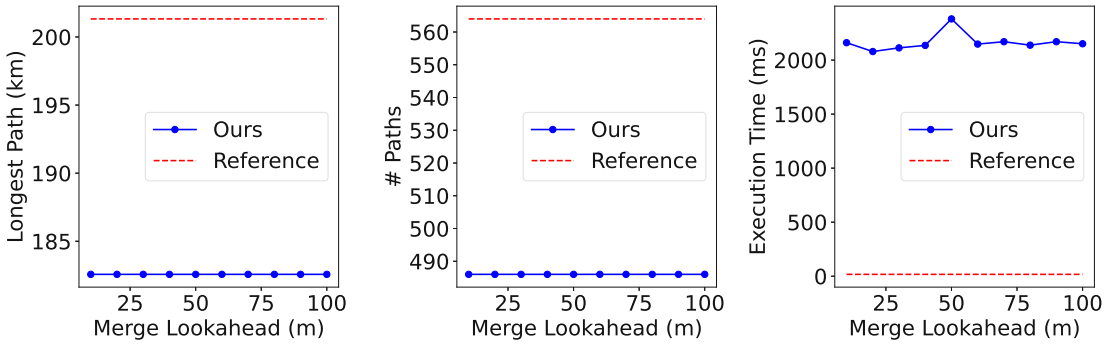


Figure 7.25: Run for Austin-Dallas (Texas) Area

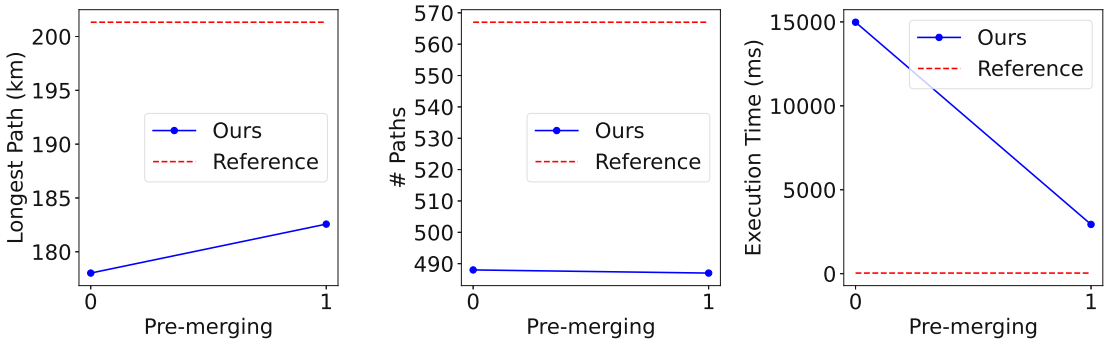


Figure 7.26: Run for Austin-Dallas (Texas) Area

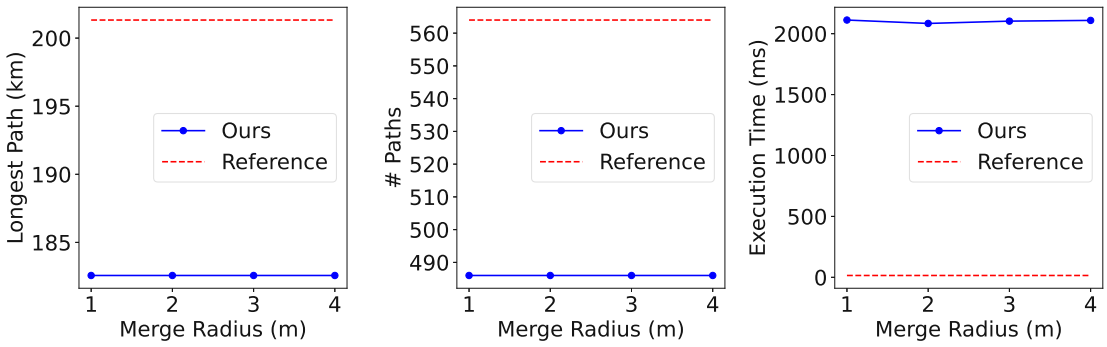


Figure 7.27: Run for Austin-Dallas (Texas) Area

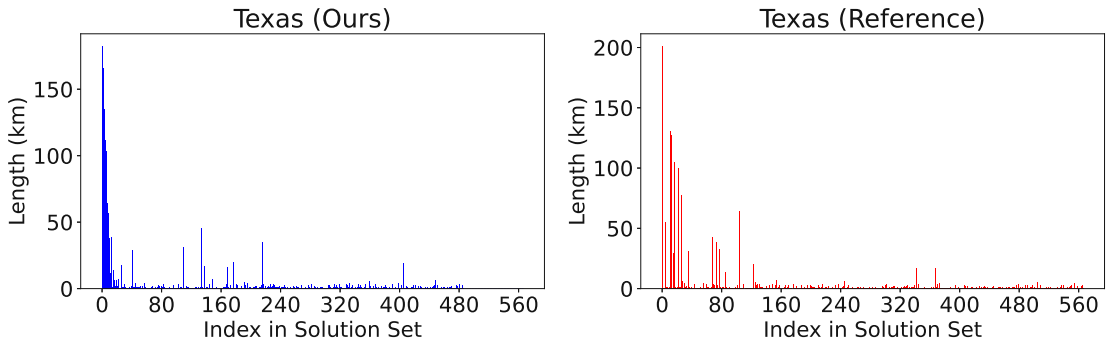


Figure 7.28: Length per Path

7.6 Case Study

Typically, one of the strongest metrics in evaluating algorithms is the accuracy of the result. In this experiment we import data of railway track geometries from OEBB. The data included in this dataset comprise the main tracks of the Austrian railway network. The data is openly available on [10] and can be downloaded in GeoJSON, shapefile or xlsx format. Even though the data can be fetched as GeoJSON with a WGS84 coordinate system, it has a different format to the data from OpenStreetMaps. Each track segment has a property with “von_id” and “nach_id” where the from_id is equal to the to_id of the previous segment. After converting the data to the uniform format we use for the track creation, we can run the algorithms on the segments provided by the OEBB dataset. Additionally, we were able to merge the OEBB segments by their id’s which should in theory result in the ground truth of data. Unfortunately, however, the OEBB data itself is inconsistent and contains some errors. In particular some intermediate segments are skipped and in some instances two segments point to the same successor segment, i.e., they have the same “nach_id”. But even considering these inconsistencies, the original dataset serves as a good reference point.

OEBB Benchmark

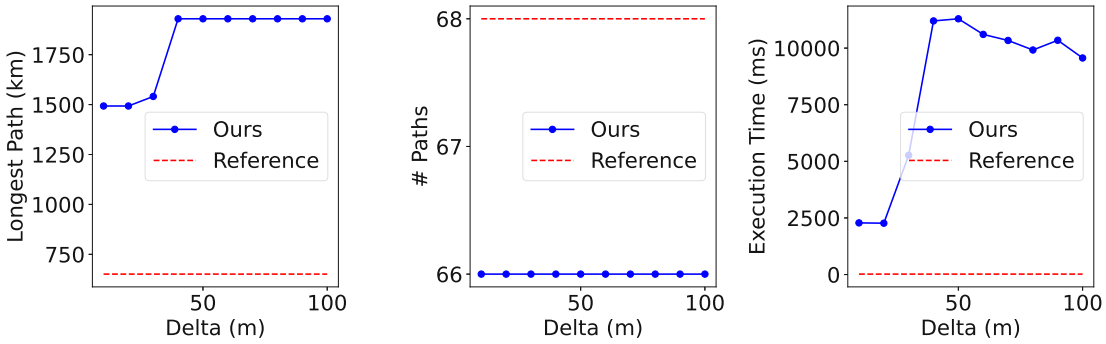


Figure 7.29: Run Oebb Track

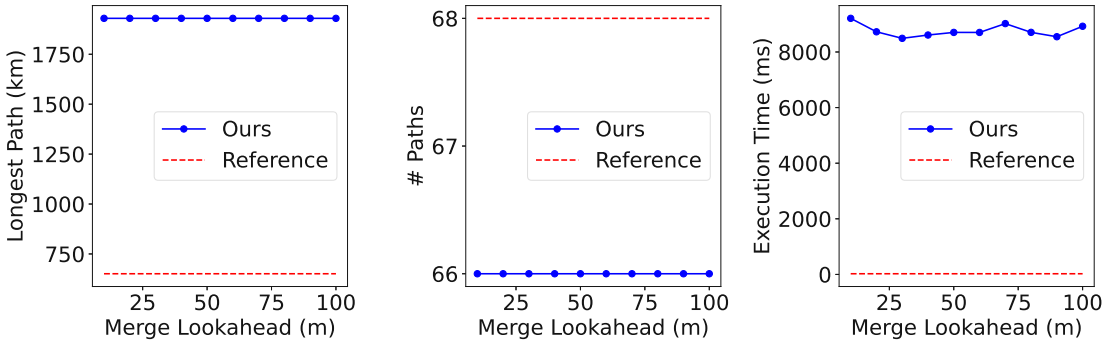


Figure 7.30: Run Oebb Track

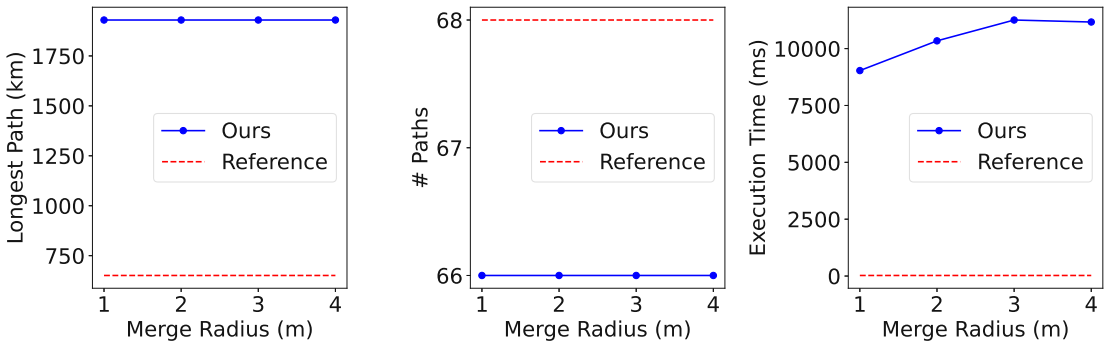


Figure 7.31: Run Oebb Track

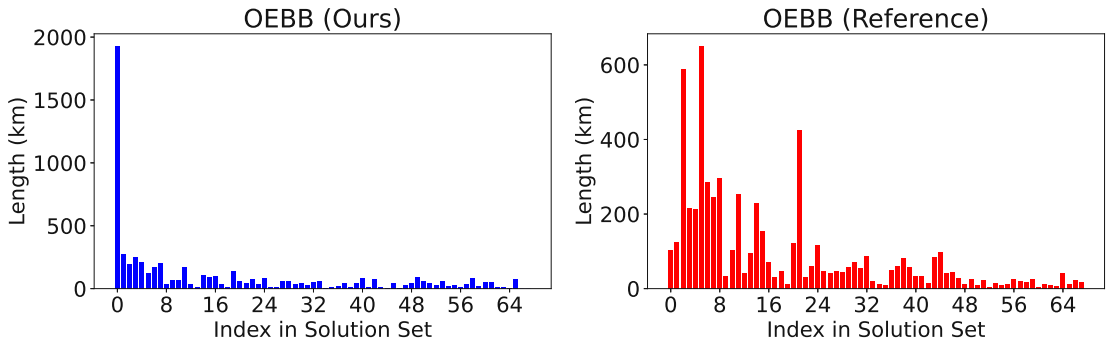


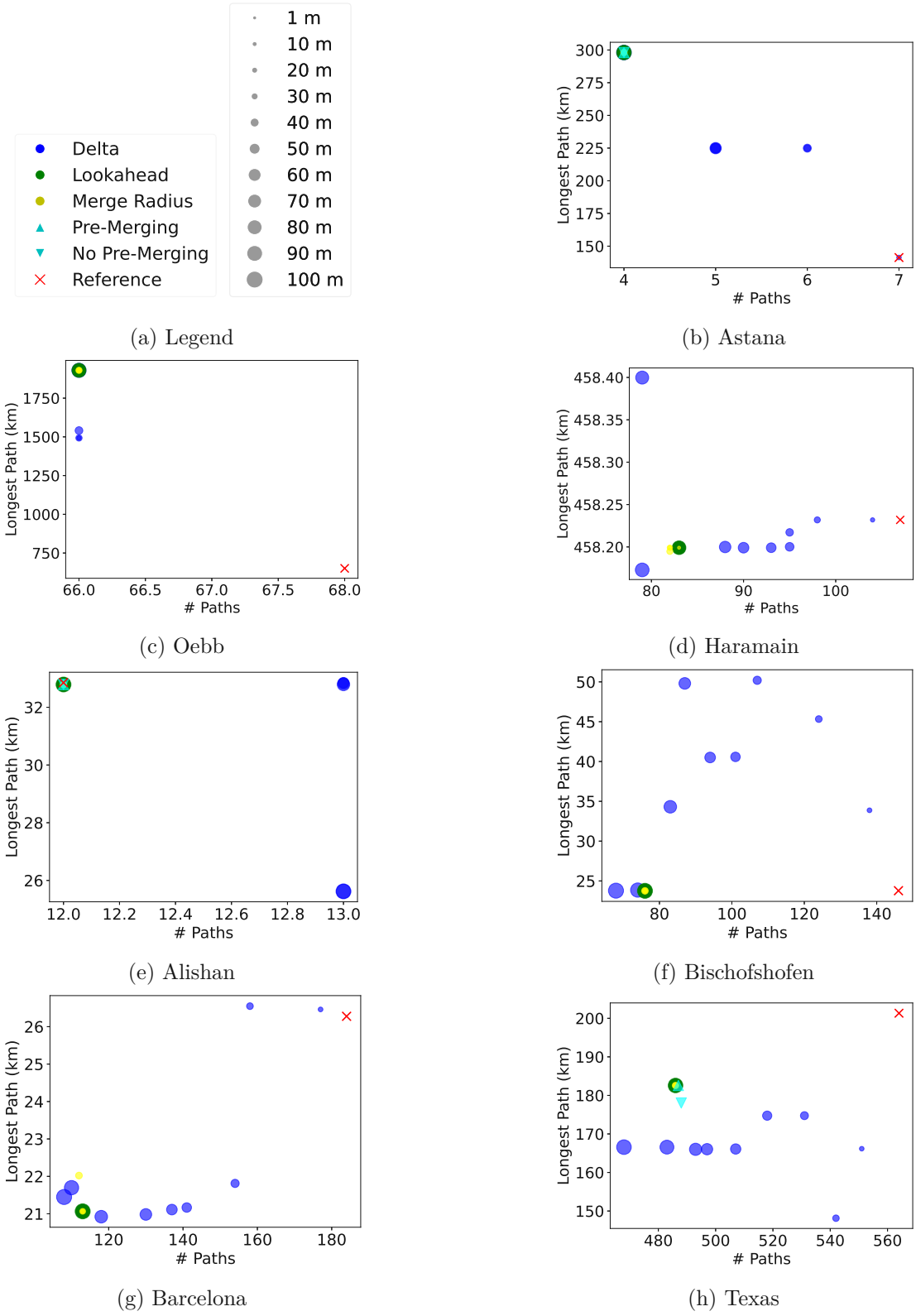
Figure 7.32: Length per Path

From the experiments of this case study we once again see that the most significant parameter is the delta value. This run further undermines the importance of the segment pre-merging, as the run without pre-merging enabled did not terminate in adequate time. Most interestingly, the longest path resolved with the new algorithm is a lot longer to that from the reference data. In the original OEBB data there are tracks which would theoretically extend further but at one point in the track a new line with new id is started. That is probably why the reference data shows such a different solution.

7.7 Combined Results

The scatterplots from Figure 7.33b through Figure 7.33h summarize all the runs with variable parameters into a single figure per input instance. The y-axis represents the length of the longest path found in that run (in kilometers), while the x-axis represents the size of the solution set. Generally, the further left on the x-axis and the higher up on the y-axis the better the result of the run. However, when interpreting results, we weight the x-axis much more since there can be larger deviations in the longest path as discussed in Section 7.4.1. The different parameters for each run are distinguished by different markers. Circles depict a distance or radius parameters, with the circle's size corresponding to the distance or radius in meters. The bigger the distance or the radius the bigger the circle in the plot. For the pre-merging parameter, which is a boolean parameter, the marker is a triangle where the triangle pointing upwards depicts pre-merging activated and the one pointing downwards pre-merging deactivated. The red 'x' denotes the reference solution. Figure 7.33a provides a legend for the scatter plots in Figure 7.33b to Figure 7.33h.

Upon inspecting the results of runs with different parameters, we can observe that the delta parameter has the most significant impact on the results. This makes sense because the larger the delta value is the bigger the amount of segments that are considered for the track creation in the algorithm. We can see that a greater delta value generally results in a smaller solution set but not always in a longer longest path. The runs performed with the new algorithm also mostly outperform the reference algorithm. Changes in



(a) Legend

(b) Astana

(c) Oebb

(d) Haramain

(e) Alishan

(f) Bischofshofen

(g) Barcelona

(h) Texas

Figure 7.33: Scatter plots for different tracks with different parameters

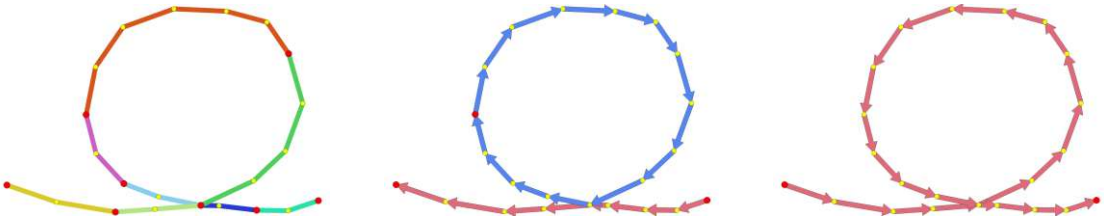
the merge radius and merge-lookahead do not impact the solution significantly in terms of output. This makes sense since we only merge segments that are unambiguously connected meaning they should eventually be put in the same solution by the track creation algorithm.

7.8 Edge Cases

To better understand the behavior of the algorithm in edge cases, we have created some artificial scenarios that represent some of the known issues that we discussed in Section 4.3 and Section 4.5. In this section we visualize the results of the greedy local fit algorithm we described in Chapter 4 and the experimental algorithm we introduced in Chapter 5 and Chapter 6 with the most promising parameter values from Section 7.7 on these scenarios. We have separated the scenarios from the benchmarking tool since the values measured in the benchmarking tool (execution time, longest path etc.) are not as significant on the small instances we present here.

7.8.1 Self intersecting track

In Figure 7.34 we run the algorithms on a track instance that self-intersects. Figure 7.34a visualizes the input data for this run. It contains eight track segments where endpoints of four distinct segments lie on the same location. The greedy local fit algorithm divides the track into two paths creating a solution set of cardinality two. One solution path being the loop creating a circle with the other one the remaining track as shown in Figure 7.34b. The experimental algorithm detects this loop and creates a single path being the favored solution in this example as shown in Figure 7.34c.

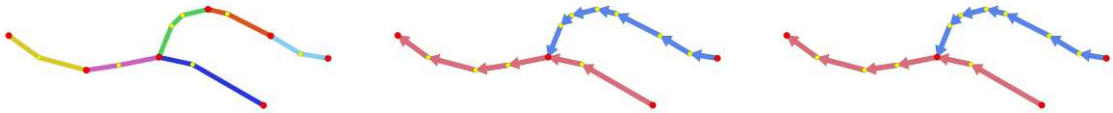


(a) Input (b) Greedy Local Fit (c) Experimental

Figure 7.34: Self intersecting track

7.8.2 Forked track

In Figure 7.35 we run the algorithms on a track instance that forks into two distinct paths. The solution generated by the different algorithms on this instance is identical. This is likely due to the trajectory of the track segments. The leftmost segments (yellow and pink) are unlikely to be part of the same track as the green brown and cyan segments as it would have low continuity.



(a) Input (b) Greedy Local Fit (c) Experimental

Figure 7.35: Forked track

7.8.3 Overlapping track segments

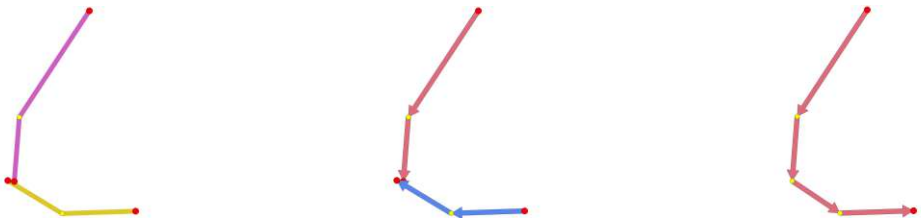
In Figure 7.36 an input instance with two overlapping track segments is provided. Here the greedy local fit algorithm fails to merge the two segments. The experimental algorithm on the other hand returns a single path for that instance.

(a) Input (b) Greedy Local Fit (c) Experimental

Figure 7.36: Overlapping track segments scenario

7.8.4 Small segment scenario 1

This scenario covers a possible issue outlined in Figure 4.3. The heuristic of the greedy local fit algorithm scores the two segments too low in comparison to the experimental algorithm.

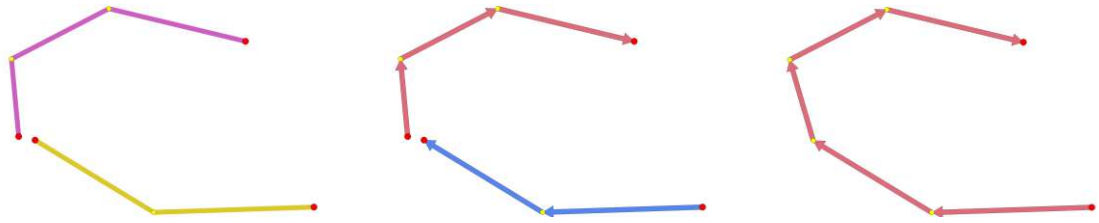


(a) Input (b) Greedy Local Fit (c) Experimental

Figure 7.37: Small segments scenario 1

7.8.5 Small segment scenario 2

This scenario also covers an issue presented in Figure 4.3. The experimental algorithm is able to detect the two segments as connected while the greedy local fit algorithm fails to do so.



(a) Input (b) Greedy Local Fit (c) Experimental

Figure 7.38: Small segments scenario 2

7.9 Summary Discussion

From the experiments conducted in this chapter we can see that the new algorithm generally finds solutions with smaller amount of paths compared to the greedy local fit algorithm. Since the new algorithm performs an exhaustive search for expanding paths, it consistently delivers superior results. The runtime, however, was also significantly higher than the greedy local fit algorithm.

In our analysis, we identified the delta parameter as the most influential factor in the algorithm's performance. On larger instances we can clearly see a rapid increase in runtime with larger delta values. A value of 80m seems like a well-balanced trade-off between runtime efficiency and solution quality. On smaller instances, the difference in runtime was marginal.

A variation in the merge radius and merge lookahead parameters had minimal impact on both the solution set and runtime. For larger instances, however, enabling pre-merging becomes essential for the experimental algorithm to terminate within reasonable timeframes.

While a longer longest path might indicate a preferable outcome, we believe that examining the length of all paths in the solution set, provides a better approach to interpreting the results.

7.10 Limitations

In addition to the input instances described in Section 7.1, we performed some runs on huge instances (covering hundreds of square kilometers) and observed that the algorithm struggles to find a solution in feasible time for them. The size of the area that the algorithm was still able to process in a reasonable time was different depending on the

density of track segments. Urban areas contain a high number of track segments in a small space, which greatly increases the number of possibilities for creating tracks. In rural areas the track segments are distributed more sparsely, with often having just one or two tracks in the surrounding area, thus enabling algorithm process larger areas in feasible time. We also observed that for some bigger but still manageable instances the algorithm with pre-merging enabled was able to find a solution in a reasonable time while the algorithm without pre-merging enabled failed to do so. These runs had to be aborted after a few hours of runtime. Therefore, the size of the input instances should be selected accordingly in order to have satisfactory results. It is suggested to filter out as many unnecessary track segments as possible while fetching data from the data providers. If possible one can perform a filtering of only main railway tracks or railway tracks that are tagged with a name or id that corresponds to the track that needs to be extracted. Additionally, only the minimal area surrounding the track should be provided as a bounding box when performing the fetching of OSM data via a bounding box query. When requesting a very large input instance, it is better to first segment the data into smaller parts and then run the algorithm on each part of the track separately.

CHAPTER 8

Conclusion

8.1 Summary

This thesis has presented a purely geometric approach on extracting single railway tracks from generic geometry data, such as the one available on geometric databases like OpenStreetMap. By addressing the challenge of correctly assigning track segments to single tracks, ordering and merging them correctly, the proposed algorithms and methodologies presented in this thesis have shown promising results in improving an existing approach (which relied on the greedy local fit algorithm presented in Chapter 4) to the problem. We discussed how to fetch and convert raw OSM data into a data structure that can be used as input for the novel experimental track creation algorithm. By introducing a weight function that takes the distance, relative distance and continuity between two segments into consideration we create a segment graph and subsequently find a maximum weighted path cover for it. We developed an exhaustive line expansion that takes all plausible concatenations of track segments into consideration, creating paths with maximum total weight.

We characterized some subsets of problematic input instances in Chapter 4 and narrowed them down into edge cases that we used for our experiments.

By implementing a benchmark framework for the algorithm we were able to run experiments semi-automatically on multiple input instances. Furthermore, the evaluation of the algorithms through experiments on those instances has provided valuable insights into the performance and solution quality of the proposed approach. In particular, a comparison to the greedy local fit algorithm, in terms of different metrics like the longest path, the size of the solution set and the runtime performance as well the length of the discovered paths, was conducted and lets us assess the benefits of the new approach in more detail. The case study on the main railway track lines from the Austrian railway system with the OEBB data provides an empirical validation of the improvements of

the newly introduced algorithm. Lastly, we created some visualization of the results in Chapter 7. The implementation of the algorithm, which was the practical part of this thesis, has already replaced the greedy local fit algorithm in the software project TrackDB of tmc.

8.2 Future Work and Outlook

While the problem solved by the algorithm presented in this thesis might be very specific, we believe that it lays a solid foundation for future research in the field of geometric data processing of infrastructure networks. The results can be extended to other infrastructure networks such as motorways and might even be useful for other geometric topics unrelated to infrastructure.

Our focus in this thesis was dedicated purely to the geometry of the track segments, and we did not investigate beyond that. It is worth noting that incorporating additional metadata from the input instances (e.g. OSM tags), to the track extraction process, could further enhance the quality of the extraction.

To further improve the quality of the heuristic algorithm, it would be a worthwhile consideration for future work to incorporate machine learning strategies to better detect two connected segments, consequently improving the local weight function. Additionally, optimizing algorithmic parameters presents an opportunity for advancement, with the exploration of genetic algorithms to iteratively determine optimal parameter configurations, thereby maximizing algorithm efficacy.

Bibliography

[1] Track machine connected. https://www.tmconnected.com/en/home/. Accessed: 2024-05-03.

[2] Geoff Boeing. Osmnx: New methods for acquiring, constructing, analyzing, and visualizing complex street networks. *Computers Environment and Urban Systems*, 65:126–139, 07 2017. DOI: 10.1016/j.compenvurbsys.2017.05.004.

[3] H. Butler, M. Daly, A. Doyle, Sean Gillies, T. Schaub, and Stefan Hagen. The GeoJSON Format. RFC 7946, August 2016. DOI: 10.17487/RFC7946.

[4] Lingji Chen and Ravi Ravichandran. A maximum weight constrained path cover algorithm for graph-based multitarget tracking. In *2015 18th International Conference on Information Fusion (Fusion)*, pages 2073–2077. IEEE, 2015.

[5] Chee-Yee Chong, Greg Castanon, Nathan Cooprider, Shozo Mori, Ravi Ravichandran, and Robert Macior. Efficient multiple hypothesis tracking by track segment graph. In *2009 12th International Conference on Information Fusion*, pages 2177–2184, 2009.

[6] Universidad de Alcalá. Line smoothing. https://geogra.uah.es/patxi/gisweb/LGmodule/LGSmoothing.htm. Accessed: 2024-05-03.

[7] Geofabrik. Osm inspector. https://wiki.openstreetmap.org/wiki/OSM_Inspector, https://tools.geofabrik.de/osmi. Accessed: 2024-05-03.

[8] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2):47–57, jun 1984. DOI: 10.1145/971697.602266.

[9] Oliver Heirich. *Localization of Trains and Mapping of Railway Tracks*. PhD thesis, Technische Universität München, July 2020.

[10] Datenquelle: ÖBB-Infrastruktur AG. https://data.oebb.at/de/datensaetze~geo-netz~. Accessed: April 20, 2024.

[11] JOSM. Josm/validator. https://wiki.openstreetmap.org/wiki/JOSM/Validator. Accessed: 2024-05-03.

[12] Alireza Karduni, Amirhassan Kermanshah, and Sybil Derrible. A protocol to convert spatial polyline data to network formats and applications to world urban road networks. *Scientific Data*, 3(1):160046, Jun 2016. DOI: 10.1038/sdata.2016.46.

[13] Kenya Kobayashi, Guohui Lin, Eiji Miyano, Toshiki Saitoh, Akira Suzuki, Tadatoshi Utashima, and Tsuyoshi Yagita. Path cover problems with length cost. In *WALCOM: Algorithms and Computation: 16th International Conference and Workshops, WALCOM 2022, Jember, Indonesia, March 24–26, 2022, Proceedings*, pages 396–408, Berlin, Heidelberg, 2022. Springer-Verlag.

[14] S.T. Leutenegger, M.A. Lopez, and J. Edgington. Str: a simple and efficient algorithm for r-tree packing. In *Proceedings 13th International Conference on Data Engineering*, pages 497–506, 1997. DOI: 10.1109/ICDE.1997.582015.

[15] Junyuan Lin and Guangpeng Ren. On optimization of 1/2-approximation path cover algorithm, 2021.

[16] LocationTech. Java topology suite (jts). `http://locationtech.github.io/jts/`, 2023. Accessed: December 27, 2023.

[17] John G Manchuk and CV Deutsch. Conversion of latitude and longitude to utm coordinates. *Paper 410, CCG Annual Report*, 11, 2009.

[18] Robert Brainerd McMaster and K Stuart Shea. Generalization in digital cartography. Association of American Geographers Washington, DC, 1992.

[19] Shlomo Moran, Ilan Newman, and Yaron Wolfstahl. Approximation algorithms for covering a graph by vertex-disjoint paths of maximum total weight. *Networks*, 20(1):55–64, 1990. DOI: 10.1002/net.3230200106.

[20] OpenStreetMap contributors. OpenStreetMap. `https://www.openstreetmap.org`. Accessed: 2024-05-03.

[21] Overpass Development Team. Overpass api. `https://dev.overpass-api.de/overpass-doc/en/`. Accessed: 2023-12-04.

[22] QGIS Development Team. Qgis geographic information system. `http://qgis.org`, 2009. Accessed: 2023-10-15.

[23] Christian Rahmig and Andreas Kluge. Digital maps for railway applications based on openstreetmap data. In *16th International IEEE Conference on Intelligent Transportation Systems (ITSC 2013)*, pages 1322–1327, 2013. DOI: 10.1109/ITSC.2013.6728414.

[24] Frederik Ramm, I Names, SS Files, F Catalogue, P Features, N Features, and C Cars. Openstreetmap data in layered gis format. *Version 0.6*, 7, 2014.

[25] Rechtsinformationssystem des Bundes. Bundesrecht konsolidiert: Gesamte rechtsvorschrift für eisenbahnbau- und -betriebsverordnung, fassung vom 27.04.2023. https://www.ris.bka.gv.at/GeltendeFassung.wxe?Abfrage=Bundesnormen&Gesetzesnummer=20006077.

[26] Fares Tabet, Sikha Pentyala, Birva H. Patel, Abdeltawab Hendawi, Peiwei Cao, Ashley Song, Harsh Govind, and Mohamed Ali. Osmrunner : A system for exploring and fixing osm connectivity. In *2021 22nd IEEE International Conference on Mobile Data Management (MDM)*, pages 193–200, 2021. DOI: 10.1109/MDM52706.2021.00039.

[27] F. Sellerhof V. Berkhahn. Anwendung der geometrischen modellierung auf trassierungen im dreidimensionalen geländemodell. *Forum Bauinformatik, Cottbus 96*, 20, 1996.

[28] A. Wolin, B. Paulson, and T. Hammond. Sort, merge, repeat: An algorithm for effectively finding corners in hand-sketched strokes. In *Proceedings of the 6th Eurographics Symposium on Sketch-Based Interfaces and Modeling*, SBIM '09, page 93–99, New York, NY, USA, 2009. Association for Computing Machinery. DOI: 10.1145/1572741.1572758.

[29] Cem Yuksel, Scott Schaefer, and John Keyser. On the parameterization of catmull-rom curves. In *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*, SPM '09, page 47–53, New York, NY, USA, 2009. Association for Computing Machinery. DOI: 10.1145/1629255.1629262.