Automated Generation of Transit Maps

Dissertation zur Erlangung des Doktorgrades der
Ingenieurwissenschaften der Technischen Fakultät der
Albert-Ludwigs-Universität Freiburg

vorgelegt von
Patrick Brosi

Albert-Ludwigs-Universität Freiburg
Technische Fakultät
Institut für Informatik

2022

Abstract

We consider the problem of automatically drawing public transit maps from schedule data. The maps may either be geographically accurate (with lines closely following their real-world course), or schematic. In particular, we study the following subproblems: Given schedule data without geographical line courses (shapes), how can these shapes be extracted from geographical datasets? Given then a collection of such shaped vehicle trips, how can we construct a topological network graph whose edge segments will not overlap in a rendering, and are labelled with the set of lines passing through them? How can we efficiently find line label permutations that will produce as few line crossings and/or separations as possible in the final map? How can we efficiently draw a schematic version of the map which may also consider obstacles, or respect the original geographical line courses to some degree?

We first describe a map-matching approach for schedule data based on a hidden Markov model and perform a quality evaluation on several schedule datasets. In this context, we also study various methods for deciding whether two public transit station identifiers are similar. Several speed-up techniques for the map-matching process are additionally presented and evaluated.

We then proceed to describe and evaluate a tailored map construction approach to build the topological network graph from such data. This approach considers line turn restrictions and clusters similar stations into single nodes.

To find optimal line label permutations, we formulate a variant of the classic Metro Line Crossing Minimization Problem (MLCM) better suited for rendering, which we call the Metro Line Node Crossing Minimization Problem (MLNCM). We give a weighted variant (MLNCM-W) and a variant which also considers line separations (MLNCM-WS), a concept we consider important for the quality of the rendered maps. We prove the NP-hardness of all four problems and describe various approximation methods, as well as an integer linear program, to solve them to optimality. To speed up solution times and to improve the approximation quality, we formulate a set of line graph transformation rules which implicitly compute an optimal partial ordering of the lines and typically break up the problem into many smaller subproblems.

In the context of schematic maps we describe a novel method for drawing octilinear transit maps. Both an integer linear program and a fast approximation method based on the repeated calculation of shortest paths in a special octilinear grid graph are presented. We evaluate several speed-up techniques and demonstrate that the method also works for other layouts (e.g. orthoradial or hexalinear).

This work also provides a practical overview of how to finally render the maps in an esthetically pleasing way. All our approaches were implemented as separate publicly available tools, forming an easily extendable pipeline.

Zusammenfassung

Wir behandeln das Problem der automatisierten Erstellung von ÖPNV-Karten aus Fahrplandaten. Die Karten können dabei sowohl geografisch korrekt, als auch schematisch sein. Insbesondere untersuchen wir folgende Teilprobleme: Wie können fehlende geografische Linienverläufe (Shapes) aus vorhandenem Kartenmaterial extrahiert werden? Wie kann aus einer Menge von Fahrten mit Shapes das darin enthaltene topologische Netzwerk so abgeleitet werden, dass Kanten mit darauf verkehrenden Linien versehen sind und sich später nicht überlappen? Wie können schnell Permutationen dieser Linien gefunden werden, so dass sie sich in der Karte so selten wie möglich kreuzen und/oder trennen? Wie können effizient schematische Karten erzeugt werden, die auch Hindernisse oder die ursprünglichen geografischen Linienverläufe berücksichtigen?

Wir beschreiben zunächst einen auf einem Hidden-Markov-Modell basierenden Map-Matching-Ansatz für Fahrplandaten und führen eine Evaluation auf mehreren Fahrplandatensätzen durch. Vor diesem Hintergrund untersuchen wir auch Methoden um zu entscheiden ob Paare von Stationen ähnlich sind. Wir untersuchen und evaluieren außerdem mehrere Methoden zur Beschleunigung des Map-Matchings.

Weiter beschreiben und evaluieren wir einen maßgeschneiderten Map-Construction-Ansatz zur Extraktion des toplogischen Netzwerks. Dieser Ansatz kann Abbiegeverbote von Linien berücksichtigen und ähnliche Stationen in einzelne Knoten überführen.

Um optimale Permutationen der Linien zu finden, formulieren wir eine Variante des klassischen Metro Line Crossing Minimization Problems (MLCM), die das Rendern der Karten vereinfacht. Wir nennen diese Variante das Metro Line Node Crossing Minimization Problem (MLNCM) und beschreiben außerdem eine gewichtete Variante (MLNCM-W) sowie eine Variante die auch Linientrennungen berücksichtigt (MLNCM-WS). Letzteres erachten wir als sehr wichtig für die Kartenqualität. Wir beweisen die NP-Härte aller vier Probleme und beschreiben sowohl Näherungsmethoden, als auch ein ganzzahliges lineares Programm (ILP) zum Finden optimaler Lösungen. Zur schnelleren Optimierung und zur Qualitätsverbesserung der Näherungsmethoden formulieren wir verschiedene Transformationsregeln, die implizit optimale partielle Ordnungen auf den Linien berechnen und das Problem typischerweise in viele kleinere Unterprobleme teilen.

Für die Schematisierung entwerfen wir eine neue Methode zum Erstellen oktilinearer Karten. Hier beschreiben wir sowohl ein ILP als auch einen schnellen Approximationsalgorithmus basierend auf der Berechnung kürzester Pfade in einem oktilinearen Gittergraphen. Wir untersuchen außerdem Beschleunigungsansätze und zeigen, dass unser Verfahren auch mit anderen Layouts (z.B. orthoradial oder hexalinear) funktioniert.

In dieser Arbeit wird außerdem ein praxisorientierter Überblick über das ästhetisch ansprechende Rendern der Karten gegeben. Alle unsere Methoden wurden in separaten und öffentlich verfügbaren Tools implementiert, die zusammen eine einfach zu erweiternde Pipeline bilden.

Acknowledgments

My sincere thanks go to Hannah Bast for her advice and guidance during the 5 years I worked on this thesis, and for all her thoughtful comments during nightly sessions before paper deadlines. I am extremely grateful for the opportunities, freedom, and time provided to me at the Chair of Algorithms and Data Structures. Special thanks go to Sabine Storandt for all her valuable insights and her support with linear programming. Both Hannah and Sabine gave many additional ideas that unfortunately did not find their way into this thesis due to time constraints.

I would also like to thank my colleagues and former master students at our chair Björn Buchhold, Thomas Goette, Matthias Hertel, Johannes Kalmbach, Theresa Klumpp, Claudius Korzen, Axel Lehmann, Markus Näther, Natalie Prange, and Niklas Schnelle for an always very pleasant working environment, frequent discussions over computer science related topics as well as real-world trivia, and cooking sessions in our kitchen. Additional thanks go to Natalie, Matthias, and Axel for their thorough proofreading. I also wish to thank our secretary Heike Hägle for her help in navigating the administrative jungle, and our administrator Frank Dal-Ri for his technical support.

Some of the anonymous reviewers of the publications preceding this work went out of their way to provide suggestions for improvement. I would also like to express my gratitude for these helpful comments.

Finally, I want to thank my family, especially my wife Manuela, for all their support and love.

Contents

Chapter 1

Introduction

Transit maps are diagrams of public transportation networks. They depict the lines and stations of e.g. subway, bus, rail, or tram systems, but may also show points of interest or geographical landmarks. Their primary function is to give passengers a network overview to help them plan their route. To this end, transit maps must give a clear indication of which stops are served by which lines. They should also respect the locations of stops as well as the geographical line course, at least to some extent.

Historically, transit maps had a high degree of geographical accuracy. In 1931, Harry Beck proposed his famous schematic design of the London subway network [68, 92] (he was not the first to use schematic maps [45]). In his design, edge segment orientations were always multiples of 45 degrees (see Figure 1.1). While there are other schematization methods used in real-world transit maps (already in 1933, a map of the Berlin S-Bahn system featured an orthoradial-like layout [82]), this so-called octilinear (or *octo*linear [151]) design has since become the de facto standard and has largely replaced geographically accurate transit maps.

Map services like OpenStreetMap, Google Maps, or Bing Maps, however, require geographically accurate transit maps which can be used as overlays over existing maps or satellite imagery. In particular, the global coverage of these map services requires an automated generation from raw schedule data, as designing and curating them by hand on a global scale is unrealistic, and schedule data is often the only dataset available. Most map services already offer such an automatically generated overlay, but the quality is often disappointing (see Figure 1.2).

For maps used in print, the automated generation of *schematic* transit maps is also relevant. Such systems may be used for the fast prototyping of network extensions, automated creation of planning documents (see Figure 1.3), or for the (assisted) creation of passenger maps. In particular, assistance systems in map editors require fast generation times. Some transit maps also feature a gradual level of schematization (see for example Figure 1.4), which an automated approach should also allow for.

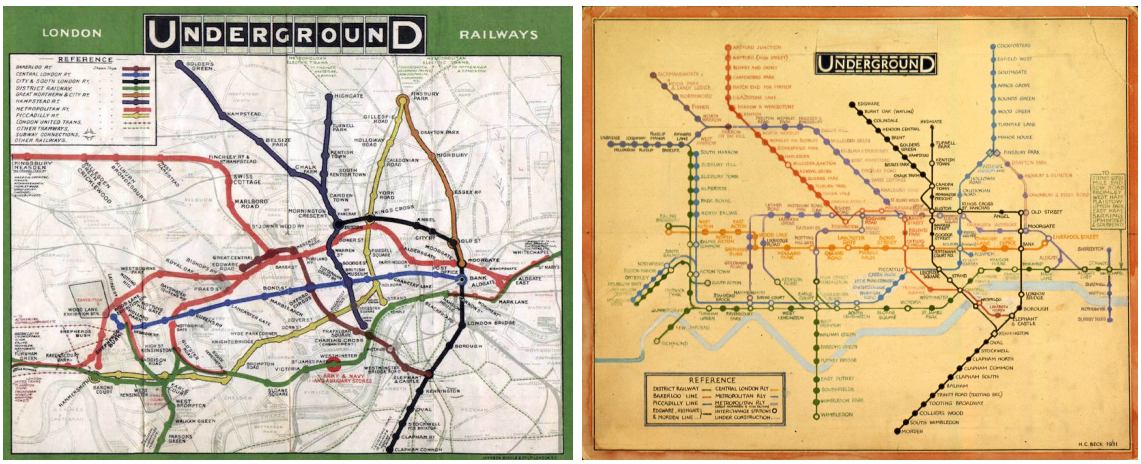


Figure 1.1: Left: 1908 map of the London subway network. Line courses have a high degree of geographical accuracy and are used as an overlay over a city map. Right: octilinear map proposed by Harry Beck in 1931. It is geographically accurate only to a limited extent. All line segments follow vertical, horizontal, or diagonal segments.

This gives rise to several algorithmic challenges: Given only schedule data (and map data describing the physical transportation network), how can the geographical line courses (shapes) be obtained efficiently, and in high quality? Given such shapes, how can we construct the underlying network graph in such a way that overlapping segments are avoided in the final map, and enough space around them is left to render the lines? How can we avoid unnecessary crossings (or separations) between lines in the final map? How can we schematize a network fast enough for the method to be used in map editors? How can the schematization process respect obstacles, and/or follow the original geographic line courses to some extent?

In this work, we address these challenges and describe a complete and efficient pipeline to automatically generate such transit maps from raw schedule data. Despite extensive research in this area over the last two decades, this problem was never addressed in its entirety. We identify and describe the key algorithmic problems of each step, briefly discuss existing approaches, describe alternative or adapted solutions where reasonable and evaluate our results on public transit networks from around the world. All our methods have been implemented in publicly available tools.

1.1 Overview

This work is organized as follows: in the remainder of this introduction, we provide a high-level overview of our pipeline and contributions (Section 1.2) and give an overview over concepts, notations and definitions used throughout this work (Section 1.3). All

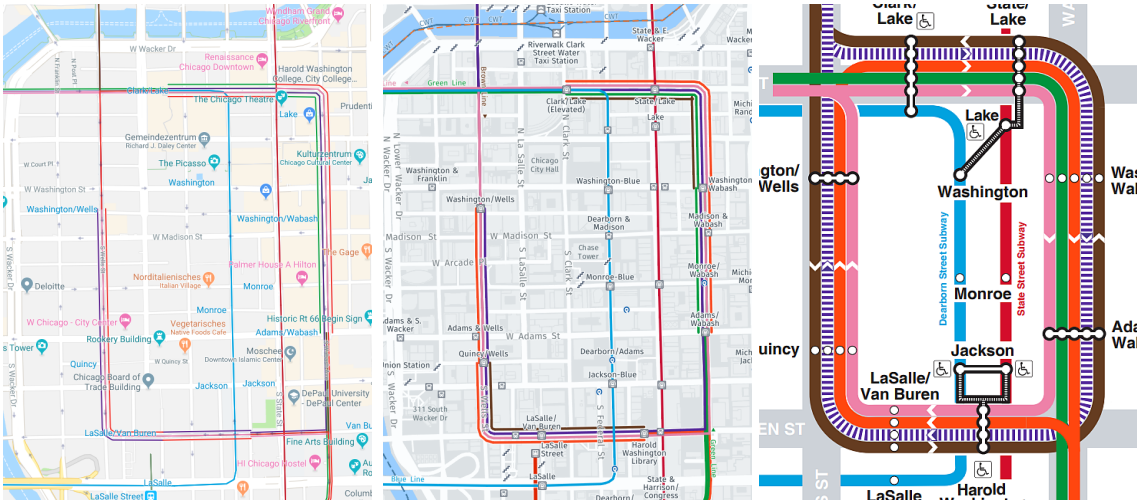


Figure 1.2: Left: The Chicago loop in the transit layer of Google Maps at the publication time of our work on geographically accurate transit maps. Lines are impossible to follow visually. Middle: The same map extent on Here maps. Right: The official Chicago Transit Authority map of the same area.

chapters are then preceded with a list of their related work, a detailed list of contributions, and an overview of the notations and concepts used only in the respective chapter.

In Chapter 2, we describe how the geographical course of a public transit vehicle can be extracted from geographical data like OpenStreetMap given only the stations (described by a *station identifier*, namely the coordinates and label) served by the vehicle. In Chapter 3, our goal is to infer the topological network graph (called the *line graph*) of a public transit system, given the geographical courses of its vehicles as input. Chapter 4 presents and evaluates various approaches for finding the optimal line ordering on individual segments in the final drawing. In Chapter 5 we then describe how to automatically find *schematic* drawings of line graphs. Chapter 6 then gives a hands-on description how the line graphs can be rendered into final maps. In this context we also discuss labeling, although we emphasize that further research is required here. Finally, Chapter 7 gives an overview over the software developed in the course of this work.

1.2 Contributions

We consider the following the main contributions of this work:

We adapt a state-of-the art approach for map-matching to work with very sparse public transit vehicle trajectories as they typically appear in real-world schedule data. In this data, trajectories are usually only sampled at stations, with straight-line geometries in between. A dual-graph representation of the underlying transportation network is

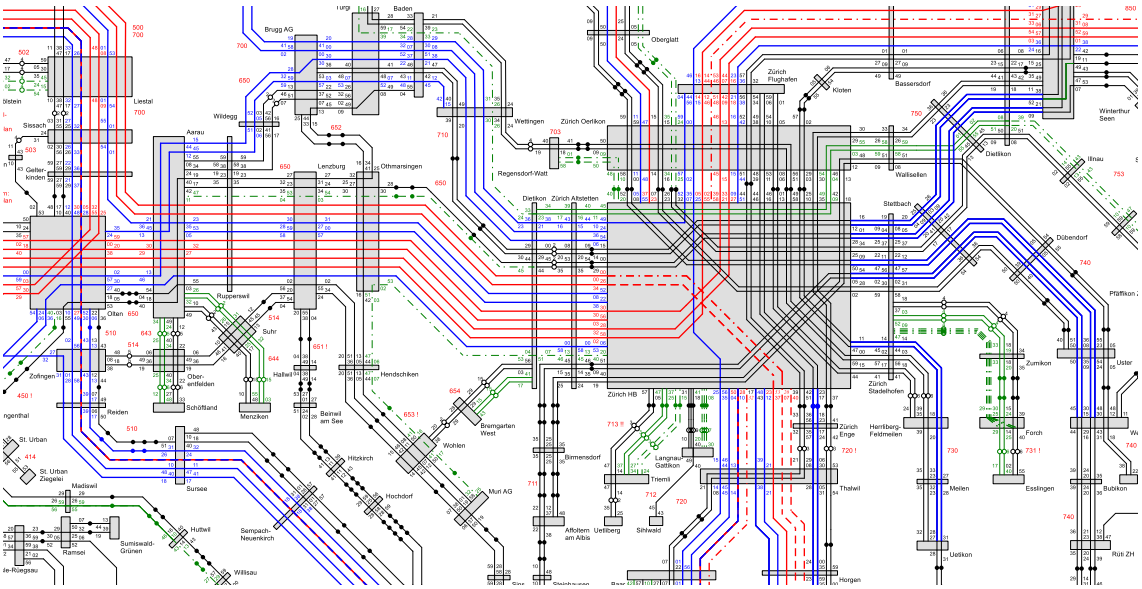


Figure 1.3: Excerpt from the official 2019 Swiss timetable map which depicts the nationwide clock-face schedule. The octilinear map is created manually each year and aimed at professional transit planers as well as the interested public.

used to include turn restrictions into the map-matching process. We motivate this with a comprehensive study of the shape quality in public schedule datasets. Several speedup techniques are also examined. Preliminary results of an earlier version of our approach have been published in the proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS 2018). The results can be explored via TRAVIC[1], a real-time visualization tool which is served by a highly efficient backend. Both TRAVIC and the backend have been published as separate works in the proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS 2014) [18, 17].

We evaluate and develop methods to check whether two station identifiers as they usually appear in schedules or geographical databases (consisting of a geographical coordinate and a label) are similar. This is a problem that arose multiple times in the context of this work. Preliminary results have been published in [15].

We describe and evaluate a method that is able to infer the network graph (the *line graph*) of a collection of vehicle trajectories. To achieve this, we develop a map construction technique which is tailored at producing line graphs fit for rendering (which we call *free* line graphs).

In the context of metro map drawing, we give a formulation of the classic Metro-Line Crossing Minimization problem (MLCM) which is better suited for rendering (MLNCM)

[1] `https://travic.app`

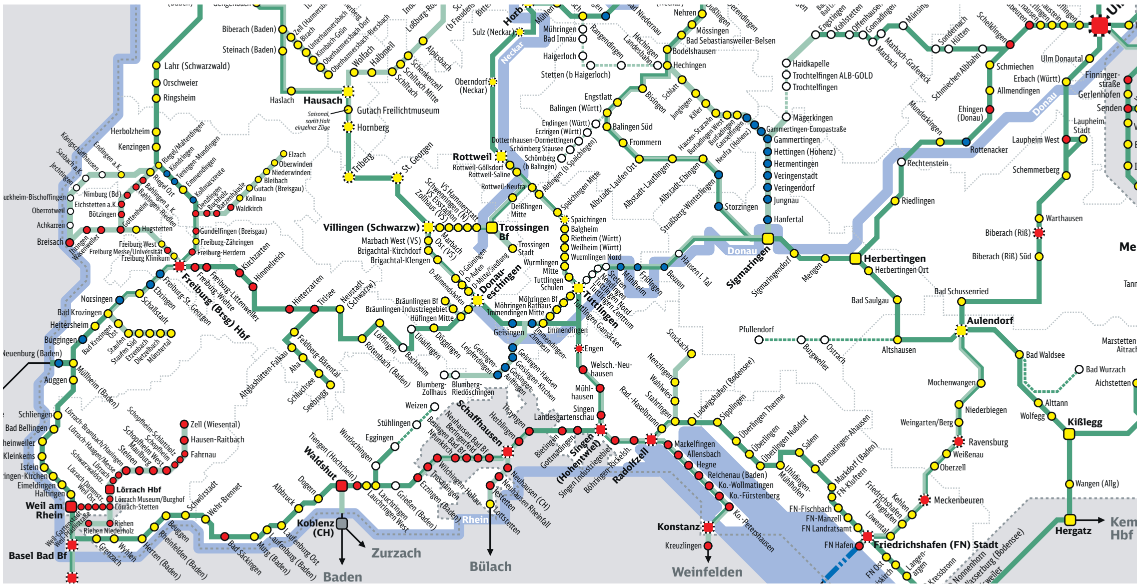


Figure 1.4: Excerpt from the official 2020 regional rail network map of southwestern Germany. Although schematic, the octilinear connections between stations roughly follow their original course (even between stations) and respect geographical boundaries on the underlying schematic map (national and regional borders, rivers, lakes).

and also give a weighted variant, then called MLNCM-W. To improve both the esthetic quality and the readability of the maps, we also introduce the concept of penalizing *line separations*. We prove the NP-hardness of all variants and give two integer linear programs (ILPs) to solve them to optimality, as well as several heuristic approaches. We conduct a large experimental evaluation on real-world datasets. The results have been previously published at the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS 2018) [19]. Out of 38 accepted full papers, the work was one of six to be considered for the best paper award.

Also in the context of MLNCM, we describe and prove the correctness of several simplification rules that may be applied to the line graph prior to optimization. These rules are able to completely solve MLNCM and its variants for certain types of input graphs, and greatly reduce the search space size of the optimization problem. The simplification rules and their effect on both the heuristic and exact solution of MLNCM have been previously published in an extended article as part of the ACM Transactions on Spatial Algorithms and Systems, Vol. 5, 2019 [20]. Using these speedup techniques, we were able to find optimal line orderings in the weighted scenario with considering line separations in under 1 second even for large real-world networks.

We give a novel method for generating octilinear transit maps by finding their optimal *image* in a special octilinear base grid graph. We describe an ILP for this optimization problem and also give a heuristic method based on the repeated calculation of shortest

paths in this grid graph. To further speed up computation times and to render maps following other layout criteria (e.g. orthoradial maps), we study and evaluate our method on several alternative base grids: *sparse* grids, which retain octilinearity but greatly reduce the number of grid nodes, and non-octilinear grids (pseudo-orthoradial grids and hexalinear grids). For small to intermediate networks, our heuristic approach typically finds an octilinear drawing in under 1 second. We also demonstrate that our method is able to produce high-quality orthoradial and hexalinear maps. The approximation method, the corresponding ILP and their evaluation have been previously presented at the 2020 Eurographics Conference on Visualization (EuroVis 2020) and published as part of the Computer Graphics Forum, Vol. 39, 2020 [21]. An extension of the method, including the formulation of sparse base grids and base grids for alternative layouts, has been presented at the 17th International Symposium on Spatial and Temporal Databases (SSTD 2021) and published as part of the conference proceedings [22]. Out of 15 full papers, the work was one of four to be considered for the best paper award.

Finally, all our methods were implemented in free open-source tools. This includes our tool *pfaedle*, a tool for map-matching schedule data given in the GTFS format, and *LOOM* (Line-Ordering Optimized Maps), an easily extendable software suite for rendering and schematizing public transit maps.

1.3 Preliminaries

This section briefly outlines a core set of concepts used throughout this work.

1.3.1 Schedule Data

Schedule data can be understood as a set of *vehicle trips*. Each trip describes the route of a single vehicle through the transportation network: which stations are served, and when. Each trip additionally has a *line* associated with it, which essentially groups multiple trips into a single service (for example, bus line 12 from Downtown to Main Station). We call this set the *Bag of Trips*.

Definition 1.1 (Bag of Trips)**.** We call a tuple $(S, \mathcal{L}, \Lambda, \mathcal{T})$ a Bag of Trips (BoT). S is a set of stations $s = (n, \pi)$, where n is the station name and $\pi \in \mathbb{R}^2$ is a station coordinate. $\mathcal{L}$ is a set of lines, Λ is a set of vehicle types (e.g. bus, rail, or streetcar). $\mathcal{T}$ is a set of vehicle trips $T = (\Sigma, \tau^{\downarrow}, \tau^{\uparrow}, l, \lambda)$, where $\Sigma = s_1, \ldots, s_n$, with $s_i \in S$, is the sequence of stops served by this vehicle, $\tau^{\downarrow} = t_1^{\downarrow}, \ldots t_n^{\downarrow}$, with $t_i^{\downarrow} \in \mathbb{N}_0^+$, is the sequence of arrival timestamps, and $\tau^{\uparrow} = t_1^{\uparrow}, \ldots, t_n^{\uparrow}$, with $t_i^{\uparrow} \in \mathbb{N}_0^+$, is the sequence of departure timestamps (a trip $T \in \mathcal{T}$ arrives at $s_i \in S$ at $t_i^{\downarrow}$ and leaves at $t_i^{\uparrow}$). $l \in \mathcal{L}$ is the *line* of the trip, $\lambda \in \Lambda$ is its vehicle type. Note that for s_0 and s_n, $t^{\downarrow} = t^{\uparrow}$.

A common file format for schedule data is the General Transit Feed Specification (GTFS) [76]. GTFS consists of several comma-separated text files which hold the set of trips (`trips.txt` for meta information like vehicle type and line, `stop_times.txt` for the stop sequences), the set of stations (`stops.txt`) and the set of lines (`routes.txt`).

Other schedule data formats exist. They are often proprietary, like the HRDF format [77] (used internally at Deutsche Bahn or the Swiss Federal Railways, among others). Others are based on national standards, like VDV 452 in Germany [139], or CIF [135] in the United Kingdom, or following international standards, like the NeTEx format in Europe [138]. They can usually be translated to GTFS with some effort.

1.3.2 Graphs

Definition 1.2 (Graph)**.** A graph is a tuple $G = (V, E)$, where V is the set of nodes, and E is the set of edges connecting them.

If edges can only be traversed in one direction, G is called a directed graph (or digraph). Otherwise, it is called an undirected graph or simply a graph. For an edge $e \in E$ between two nodes $u, v \in V$, we write $e = (u, v)$ if G is directed and $e = \{u, v\}$ if G is undirected. For directed edges, we denote by $\text{tail}(u, v) = u$ the *tail* of edge (u, v) and by $\text{head}(u, v) = v$ its *head*. An undirected graph $G = (V, E)$ can be modeled with a directed graph G' by adding edges (u, v) and (v, u) to G' for each edge $\{u, v\} \in E$.

Definition 1.3 (Vertex Degree and Adjacency)**.** For undirected graphs, we say nodes u and v are adjacent if $\{u, v\} \in E$ and write $u \in \text{adj}(v)$ and $v \in \text{adj}(u)$. We denote by the *degree* $\deg(u)$ the number of nodes adjacent to u. For directed graphs, we say node v is adjacent to u if $(u, v) \in E$ (but not if only $(v, u) \in E$) and write $v \in \text{adj}(u) = \text{adj}^+(u)$. We denote by the *out-degree* $\deg^+(u)$ the number of nodes adjacent to u and define $\deg(u) = \deg^+(u)$. Additionally, we define the *in-degree* $\deg^-(u)$ as the number of nodes u is adjacent to and denote these nodes by $\text{adj}^-(u)$.

Definition 1.4 (Node and Edge Labeling)**.** Given a graph $G = (V, E)$, a node labeling is a function $V \mapsto L$ which assigns each node v a label $l \in L$. Analogously, an edge labeling is a function $E \mapsto L'$ which assigns each node v a label $l' \in L'$.

Transportation Networks

We call a directed graph $\mathcal{N} = (V_\mathcal{N}, E_\mathcal{N})$, embedded in $\mathbb{R}^2$, and modeling the physical network for a distinct vehicle type (for example, the road network for buses, the rail network for trains, the light rail network for trams, or ferry routes) a *transportation network* for that vehicle type. In particular, a label function $\Pi : V_\mathcal{N} \mapsto \mathbb{R}^2$ embeds each node on a projection of the globe (we ignore elevation), and a label function $P : E_\mathcal{N} \mapsto (p_1, \ldots, p_n)$ embeds each edge using a polyline $p_1, \ldots, p_n$. In this work, these coordinates

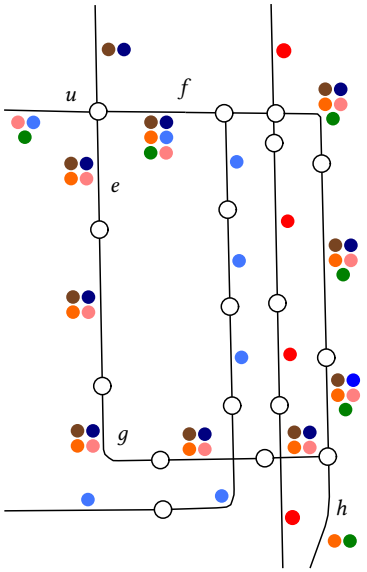


Figure 1.5: Excerpt from a linegraph $G = (V, E, L, \mathcal{L})$ for the Chicago light rail network (corresponding to the Chicago loop shown in Figure 1.2). A labeling function L assigns each edge a set of lines, with $L(e) \subseteq \mathcal{L}$, depicted here as colored circles (here, $\mathcal{L} = \{$●●●●$\}$, and e.g. $L(e) = \{$●●$\}$). In this example, edges are additionally labeled by a polyline describing the edge course, see for example edges g and h. Nodes are usually - but not necessarily - stations. For example, u does not correspond to a station (compare Figure 1.2).

are assumed to be in the Web Mercator projection (EPSG:3857), although we often give them as WGS 84 latitude/longitude pairs. For example, in a network for buses, the edges E_N correspond to road segments, and the nodes V_N correspond to intersections and/or road anchor points. The graph is directed to model unidirectional segments (e.g. one-way streets).

A node subset $\mathcal{S} \subseteq V_N$ is maintained which describes the *stations* of the network. A vertex labeling function $N(v)$ assigns each node $v \in \mathcal{S}$ a set of station labels (e.g. 'London St Pancras' and 'St Pancras Station').

Line Graphs

The fundamental structure used in this work is a *line graph*, which can be understood as the skeleton of a public transit map. Note that "line" here refers to a public transit line. In graph theory, a graph $\mathfrak{L}(G)$ that is constructed by converting each edge in G into a node in $\mathfrak{L}(G)$ is also often called a line graph [80]. In the context of this work, we will call such a graph $\mathfrak{L}(G)$ the *edge-to-node dual* of G. It will be introduced in more detail in Section 2.6.4.

In its most basic form, a line graph $G = (V, E, L, \mathcal{L})$ is an undirected labeled graph with nodes V and edges E embedded in $\mathbb{R}^2$. L is an edge labeling function which assigns an edge $e \in E$ a set of *lines* $L(e) \subseteq \mathcal{L}$, where $\mathcal{L}$ is the set of all lines. A set $S \subseteq V$ describes *station nodes*, and the $V \setminus S$ are called non-station nodes. Station nodes correspond to stations in the final map, whereas non-station nodes are typically places where lines branch. Just as for transportation networks, we often assume a label function n which assigns each node $s \in S$ a *station label*.

We also often use a function R which assigns pairs (e, f) of adjacent edges and some

line $l \in L(e) \cap L(f)$ a *line turn restriction*. That is, we mark by $R(l) = (e, f)$ that the line l does not continue from e to f (in the context of this work, we ignore the case where a turn restriction involves more than 2 edges). This will be described in more detail in Chapter 3.

1.3.3 Shortest Paths

Definition 1.5 (Edge Weights)**.** We call a graph G *weighted* if G has an edge labeling $w : E \mapsto \mathbb{R}_0^+$ (we are not interested in negative weights).

Definition 1.6 (Path)**.** A sequence of edges $e_1, e_2, \ldots, e_n$ is called a path if there is a corresponding sequence of nodes $v_1, v_2, \ldots, v_{n+1}$ such that $e_i = (v_i, v_{i+1})$. A path p is called *simple* if the sequence of nodes is distinct (this implies that a circular path is not simple). A single vertex v is a simple path. We denote by $|p|$ the number of edges in the path.

Definition 1.7 (Path Cost)**.** If p is a path in a weighted graph, it has a *cost* $c(p) = \sum_{i=1}^{n} w(e_i)$. The cost of a single-node path is 0.

Definition 1.8 (Shortest Path)**.** A path p between nodes s and t in a graph G is a shortest path if no other path p' exists with $c(p') < c(p)$.

Lemma 1.1. *A shortest path in a graph with edge weights* > 0 *is simple.*

Proof. Assume there is a shortest path p from s to t which visits a node v twice. If we remove the cycle from v to v, the resulting path p' still goes from s to t. Because $w(e) > 0$, $c(p') < c(p)$, a contradiction. □

Dijkstra's Algorithm Given a directed graph $G = (V, E)$ with non-negative edge weights and some start node $s \in V$, the shortest path from s to all nodes $V \setminus \{s\}$ can be computed using Dijkstra's algorithm [50]. The algorithm can be considered an iterative implementation of a dynamic programming approach [131] based on the following intuition: the shortest path from s to some node t is the shortest path from s to a node $v \in \text{adj}^-(t)$ (with cost c_v), plus the edge (v, t). Under all nodes $\text{adj}^-(t)$ this is precisely the node v for which $c_v + w((v, t))$ is smallest. This immediately gives rise to a recursive computation scheme in which the subproblems show great overlap (we only have to compute the shortest path to some particular node w once, but it may appear very often during the recursive computation). In the classic iterative implementation of Dijkstra's Algorithm, each node v is initially assigned a distance label $c_v = \infty$, except the start node s, which is labeled with $c_s = 0$. A priority queue Q is maintained that holds all nodes with their distance labels so far, initially only containing s with $c_s = 0$. The algorithm then repeatedly takes the node v with *lowest* c_v from Q and updates the labels

of each adjacent node u with $c'_u = c_v + w((v,u))$ if $c'_u < c_u$. v is said to be *expanded* and u is said to be *relaxed*. An important property is that each time a node v is extracted from Q, the cost label c_v is the cost of the shortest path from s to v. The extracted node is hence called *settled*. A pointer to the predecessor node from which the last relaxation came from may be maintained, allowing the later reconstruction of the shortest path.

A typical variant of Dijkstra's algorithm is to search for the shortest path from a single source node s to a single target node t, and in this case, the search can be aborted if t is settled (or if the priority queue runs empty, in which case there is no path from s to t). If a Fibonacci heap is used to implement the priority queue [64], Dijkstra's algorithm runs in time $O(|E| + |V| \log |V|)$. No asymptotically faster algorithm for computing the shortest path on general graphs is known. There are, however, many established speedup methods used in practice. They range from efficient implementation techniques of Dijkstra's algorithm, over heuristics which prefer certain nodes during the expansion process and thus steer the shortest path search towards the goal [141], to elaborate preprocessing schemes which later speed up the shortest path calculation, sometimes dramatically. A typical example for an implementation-based speedup technique is to use a special monotone priority queue (for example a radix heap [3]), or to perform the search from both s and t simultaneously until the search barriers meet (Bidirectional Dijkstra). A typical example for a heuristic approach is the A^* algorithm [81], which can be considered an extension of Dijkstra's algorithm. Some well-known examples for preprocessing schemes are Landmark heuristics (in which a special A^* heuristic is precomputed [74]) and Contraction Hierarchies [70]. We only describe A^* in detail here, as it will be used several times in the following chapters. For a thorough survey on various speedup techniques for finding shortest paths in transportation networks, we refer to [23].

A^* Algorithm Instead of adding a node v with its label c_v as described above as a key to the priority queue Q, we instead use $c_v + h(v)$, where $h(v)$ is a *heuristic function* which estimates the cost from v to the target node, thus preferring nodes which are heuristically closer to the target during the expansion process. Obviously, if $h(v) = 0$ for each $v \in V$, this is exactly Dijkstra's algorithm. There are two important criteria the heuristic function can meet and which are often confusingly mixed: *(1) admissibility* and *(2) consistency*. A heuristic function h is said to be *admissible* if $\forall v \in V : h(v) \leq c(v,t)$, where $c(v,t)$ is the shortest path cost from v to t. In other words, h must never overestimate the actual cost to the target. A heuristic function is said to be *consistent* if for every adjacent pair of nodes v, u it holds that $h(v) \leq w((v,u)) + h(u)$ (an equivalent, but harder to prove requirement is $h(v) \leq c(v,u) + h(u)$, where $c(v,u)$ is again the shortest path cost from v to u). In other words, h must never overestimate the cost to the target in such a way that if we take the edge cost to the adjacent neighbor u into account, u would provide a better overall estimate. A consistent heuristic is always admissible, but an admissible heuristic is not necessarily consistent.

If h is consistent, then at the point of expanding v, the cost label c_v is the cost of the shortest path from s to v, just like with the classic Dijkstra's algorithm (see below for a proof). If h is *not* consistent, but only admissible, this is not guaranteed anymore. A node might thus require multiple expansions.

If the implementation allows such a repeated expansion of nodes, then A^* returns the shortest path from s to t with a heuristic that is admissible, but not consistent. Importantly, the target will *never* require multiple expansion with an admissible, but inconsistent heuristic, and the algorithm can hence be terminated as soon as t is expanded:

Lemma 1.2. *Given an inconsistent, but admissible heuristic h. If we expand the target node t, c_t is guaranteed to be the cost of the shortest path from s to t, and we have thus found this shortest path.*

Proof. As we have extracted t with label c_t from Q, there is no node v with $c_v + h(v) < c_t + h(t) = c_t$. Let c be the cost of the actual shortest path from s to t, and assume c_t was not optimal, that is $c_t > c$. Then there is some node $v \in V$ (this might be s itself!) with $c_v + c(v,t) < c_t$, where $c(v,t)$ is the shortest path cost from v to t. But as h is admissible, $h(v) \leq c(v,t)$, and thus $c_v + h(v) < c_t + h(t)$, a contradiction. □

Crucially, with an inconsistent but admissible heuristic, this proof only works for the target node t. For a consistent heuristic, it works for any node u: if u is expanded, the existence of some node v with $c_v + c(v,u) < c_u$ together with the consistency requirement $h(v) \leq c(v,u) + h(u)$ would imply $c_v + h(v) < c_u + h(u)$, which again leads to a contradiction. c_u was thus the cost of the shortest path to u.

To summarize: With a consistent heuristic, A^* produces the shortest path from s to t and only expands each node at most once. With an inconsistent, but admissible heuristic, A^* might require nodes to be expanded multiple times, but if t is reached, we are guaranteed to have found the shortest path.

One-To-Set and Set-to-Set Shortest Paths A simple variation of Dijkstra's algorithm may be used to compute the shortest path from a single source s into a set of target nodes T (that is the shortest path from s to *any* of the nodes in T): we simply abort the algorithm as soon as a node $t \in T$ is settled, or if the priority queue runs empty (in which case, again, there is no path from s to T).

Another variant for which the correctness is not immediately obvious are set-to-set shortest paths. Here we are interested in finding the shortest path from any node in a source node set S to any node in a target node set T.

Lemma 1.3. *If the priority queue Q in Dijkstra's algorithm is initialized with a set of nodes S, each with cost label 0, the algorithm will produce the shortest paths from* any *node in S.*

Proof. We add a virtual node v^* and connect v^* to each node $s \in S$ with an edge of weight zero. The shortest path p from v^* to any other node will contain a node $s \in S$, and the $c(p)$ will be the cost of the shortest path starting from this node s (as the initial edge from v^* has weight 0). This will be the node from S from which the shortest path starts: if this would not be the case, there would be a path p' from v^* over a different node $s' \in S$ with $c(p') < c(p)$, but p was already the shortest path. Adding all nodes S to Q with cost label 0 is exactly the state of Q after the first relaxation of all adjacent edges to v^*, and the initialization thus implicitly constructs v^* and its edges to S. □

One-to-Many Shortest Paths The one-to-many shortest path problem searches for the shortest paths (or decides that a path does not exist) from a single source node s to *all* nodes in a target set T. It is evident from the algorithm description above that a standard Dijkstra implementation already computes a superset of the desired shortest paths and we can simply stop the algorithm as soon as all nodes $t \in T$ have been settled.

Set-to-Many Shortest Paths Similar to the set-to-set shortest path problem is the set-to-many shortest path problem, which searches for the shortest path from *any* source node $s \in S$ to *all* nodes in a target set T. Obviously, the same technique as for the set-to-set case can be used, but instead of aborting the search as soon as a node in S is settled, we let it run until all nodes in S are settled, or until the priority queue runs empty.

Many-to-Many Shortest Paths Finally, the many-to-many shortest path problem searches for all pairwise shortest paths from a set of source nodes S to a set of target nodes T. A naive implementation would compute these paths by $|S| \cdot |T|$ Dijkstra runs. A more efficient way is to compute $|S|$ one-to-many shortest paths from a node s_i to $|T|$.

1.3.4 Optimization

Finding the shortest path through a graph is an optimization problem: we want to find the path that optimizes the cost function. In general, an optimization problem searches for the best solution from some solution space Ω. This solution space is often called the set of *feasible* solutions, or *feasible set.* The set of feasible solutions is defined by a set of constraints. A feasible solution (consisting of possibly multiple values $x_1, \ldots, x_n$) $\mathbf{x} \in \Omega$ is considered *better* than a feasible solution $\mathbf{y} \in \Omega$ if the *objective function* $f : \Omega \mapsto \mathbb{R}$ gives $f(\mathbf{x}) < f(\mathbf{y})$. The objective function f is also often called *loss function*, *cost function*, or *target function.* The optimization problem may also be called a *minimization problem*, as the goal is to minimize f. Each minimization problem can be transformed into a *maximization problem* (and vice versa) by using an objective function $f' = -f$.

Local Search

Local search is a widely used class of heuristic optimization algorithms. For feasible solutions $\mathbf{x} \in \Omega$, a *neighborhood* $N(\mathbf{x}) \subset \Omega$ is defined which consists of all feasible solutions $\mathbf{x}'$ which can be obtained by making a small change to $\mathbf{x}$. The exact definition of the neighborhood is highly domain specific. Given the current solution $\mathbf{s} \in \Omega$, all $\mathbf{s}' \in N(\mathbf{s})$ are then evaluated in terms of their objective function, and one of them is chosen as the new solution candidate. How this new solution candidate is selected is the distinguishing feature of these algorithms. For example, if from all $N(\mathbf{s})$ the $\mathbf{s}'$ which gives the greatest improvement is chosen, this is called *Steepest-Ascent Hill Climbing*. *Simple Hill Climbing* always selects the first better neighbor found, and *Stochastic Hill Climbing* examines random neighbors until a better one is found. Such greedy local search variants typically get stuck in a local optimum. Some variants like *Simulated Annealing* [120] hence introduce chaos into the system to push the search out of local optima.

Linear Programming

If both the objective function f and the constraints of an optimization problem are linear, the optimization problem is called *linear programming* (LP). The objective function can then be expressed as $f(\mathbf{x}) = \mathbf{c}^T\mathbf{x}$, where $\mathbf{c}$ is a vector of coefficients. The constraints can be expressed by a vector $\mathbf{b}$ of coefficients (the right-hand side of the constraints), and a matrix $\mathbf{A}$ of coefficients (the left-hand side of the constraints, each row holds one coefficient for each solution variable).

Definition 1.9 (Canonical Linear Program)**.** Given $\mathbf{c}$, $\mathbf{b}$, and $\mathbf{A}$:

$$\max \left\{ \mathbf{c}^T\mathbf{x} \mid \mathbf{A}\mathbf{x} \leq \mathbf{b} \wedge \mathbf{x} \geq 0 \right\}. \tag{1.1}$$

While the canonical form seems to restrict linear programs to maximization problems with non-negative variables and only less-than-or-equal constraints, minimization problems, negative variables, greater-than-or-equal constraints, or equality constraints can all be expressed in it.

As mentioned above, any minimization problem can be turned into a maximization problem by multiplying the coefficients of the objective function $\mathbf{c}$ by -1. Similarly, to get greater-than-or-equal constraints, the constraint can be multiplied by -1. Equality constraints of the form $a_i x = b_i$ can be modeled by two constraints $a_i x \leq b_i$ and $a_i x \geq b_i$. Finally, a free (positive or negative) variable x_i can be modeled by two non-negative variables x_i' and x_i'' and substituting x_i by $x_i' - x_i''$.

In its standard form, the unknown variables of a linear program are real numbers, that is $x_i \in \mathbb{R}$. There are several forms with additional constraints on the variables:

Definition 1.10 (Integer Linear Programming)**.** If the unknown variables of an LP are integers, the problem is called integer linear programming, and a problem instance is called an integer linear program (ILP). Solving an ILP without the integrality constraints is called LP-relaxation and may be used to compute a lower or upper bound for the target function.

Definition 1.11 (Binary Programming)**.** If the unknown variables are additionally required to be binary, the problem is called binary integer programming (BIP) (it is easy to see that a BIP can be modeled via an ILP by adding additional constraints to all integers which restrict their values to 0 and 1).

Definition 1.12 (Mixed Integer Programming)**.** If the unknown variables can be either integers or real numbers, the problem is called mixed integer programming (MIP).

While polynomial algorithms for solving linear programs exist (the first was described by Khachiyan in [93], the well-known Simplex algorithm by Dantzig [43] has exponential worst-case running time [95]), the optimization of integer linear programs, binary programs and mixed integer programs are all NP-hard [67]. In practice, ILPs are usually optimized by special solvers. Their performance has typically been optimized for decades, and they employ a wide set of sophisticated heuristics. Prominent examples (in ascending order of general performance) are the GNU Linear Programming Kit (GLPK)[2], COIN-OR CoinMP[3], and gurobi[4]. Both GLPK and CoinMP are free to use.

[2] https://www.gnu.org/software/glpk/
[3] https://www.coin-or.org/projects/
[4] https://www.gurobi.com/

Chapter 2

Map-Matching Schedule Data

Map-matching is the process of matching a sequence of sample points (e.g. GPS traces) to a geographical model of the real world (e.g. a road network). Map-matching *schedule data* means finding the most likely path a public transportation vehicle takes on a scheduled trip through its transportation network, given only the vehicle stops (consisting of a geographical coordinate and a station label) as sample points (Figure 2.1). These paths are often called *shapes*. Formally, this chapter considers the following problem:

Problem 1 (Map-Matching Schedule Data)**.** *Given a Bag of Trips* $(S, \mathcal{L}, \Lambda, \mathcal{T})$ *and transportation networks* $\mathcal{N}_\lambda$ *for each vehicle type* $\lambda \in \Lambda$*, find the most likely path* p *(its* shape*) through the respective* $\mathcal{N}_V$ *for each trip* $T \in \mathcal{T}$*. For each* T*,* p *must consist of* $|\Sigma| - 1$ *consecutive parts* $p_1, p_2, \ldots, p_{|\Sigma|-1}$*, where* p_i *is the path from stop* s_i *to stop* s_{i+1}*.*

Given the high relevance of this problem for static or dynamic visualization of public transit systems or for real-time map-matching of mobile devices to moving trains, map-matching of schedule data has received only little research attention so far. In particular, if we consider the high spatial sparseness of the sample points (the station coordinates), it is not clear at all whether existing approaches for map-matching (typically dense) GPS traces to road networks are applicable.

After a motivation and some preliminaries, we provide an overview over related work. Section 2.4 then describes several baseline approaches to this problem, followed by a description of a hidden Markov model (HMM) to find the most likely path (Section 2.5). This model only uses the spatial information contained in the schedule data. We show how the edge-to-node dual graph of the transportation network can be used to naturally include turn restrictions both for individual transitions, and between transitions in Section 2.6. Section 2.7 describes how the map-matching results can be improved by incorporating additional metadata from the transportation network. Here, we also describe methods to decide whether two station identifiers describe the same real-world station. Section 2.9 then outlines several speedup techniques. We close with an experimental comparison of our approaches in Section 2.10.

InterCity 729					
	Station	Arr.	Dep.	Lat.	Lon.
1	Genève Aero.	—	16:32	46.231	6.111
2	Genève	16:42	16:44	46.211	6.142
3	Lausanne	17:20	17:20	46.517	6.629
4	Fribourg	18:04	18:04	46.803	7.151
5	Bern	18:32	18:33	46.949	7.438
6	Zürich	19:33	19:36	47.378	8.539
7	Zürich Flugh.	19:44	19:45	47.451	8.562
8	Winterthur	19:59	20:00	47.501	8.723
9	St. Gallen	20:35	—	47.423	9.369

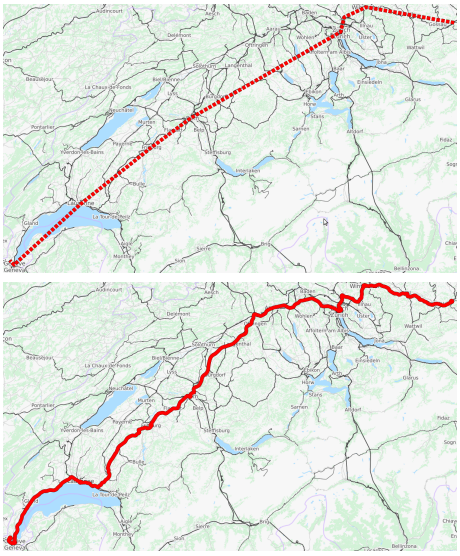


Figure 2.1: Left: Schedule data for a single trip through Switzerland. Top right: Shape obtained by the station coordinates. Bottom right: Shape map-matched by our approach.

2.1 Motivation

Our main motivation for Problem 1 is the generation of geographically accurate transit maps as described in Chapter 1. However, there are many other applications, which we briefly discuss in this section. As an additional motivation, we also conducted a survey of the general availability and quality of shapes in publicly available schedule data. The methodology and results are presented in Section 2.1.2.

2.1.1 Applications

Apart from the need for high-quality shapes in geographically accurate transit maps, Problem 1 is an important preprocessing step for many other areas which use public transit data. For example, exact shape data is also valuable for *schematic* maps, as it allows the identification of segments shared by multiple lines.

Route-planning systems for public transportation typically also visualize the calculated route on a map. For this visualization to be helpful, the shapes of the individual vehicle trips which make up the route must be known. Figure 2.2 gives three examples from the route planners of Google Maps, Deutsche Bahn, and Bing Maps where shape data was missing. The routes (from Freiburg to Berlin) are displayed with straight lines between stations and offer only little additional value. Some public transit agencies already offer interactive maps which show the live position of each vehicle in their network, and there are several such maps covering more than one agency or entire countries (e.g. the

Figure 2.2: Visualization of public transit routes in Google Maps (left), the official Deutsche Bahn route planning service (middle) and Bing Maps (right).

MTA Live Subway for New York[1], or our own work TRAVIC[2]). As real-time public transit data often only provides delay times, the vehicle shapes must be known to determine the current position. Even if exact vehicle positions are delivered, the knowledge of the vehicle shapes is advantageous. Consider, for example, a stream of live vehicle positions delivered by a GPS device on a moving bus. The positions may only be updated every few seconds or fail completely if the vehicle enters a tunnel. To ensure a smooth visualization in situations like this, the scheduled vehicle path is valuable.

Another interesting problem for which exact shape information is important is that of matching mobile devices to moving trains. A typical scenario is a smartphone app showing information regarding the train the owner is currently sitting it, but it may also be used to crowdsource delay data.

Lastly, map-matching of schedule data may be used to improve the quality of the network data itself. We can very easily use the map-matched paths to augment the original edge in the transportation network with information about the vehicle trip. For example, OSM offers the concept of public transit line relations to map line courses[3]. These may be filled automatically using this approach.

2.1.2 A Survey on Shape Availability and Quality

To further motivate our work, we conducted an extensive survey of the shape quality of publicly available schedule data. With over 900 published feeds in 2020 (see Table 2.1), `https://transitfeeds.com` is the largest collection of public transportation schedules we know of. The schedules are published as GTFS feeds. All feeds are regularly updated and archived. The archive can be queried using a simple API.

[1] `https://map.mta.info`
[2] `https://travic.app`
[3] `https://wiki.openstreetmap.org/wiki/Public_transport`

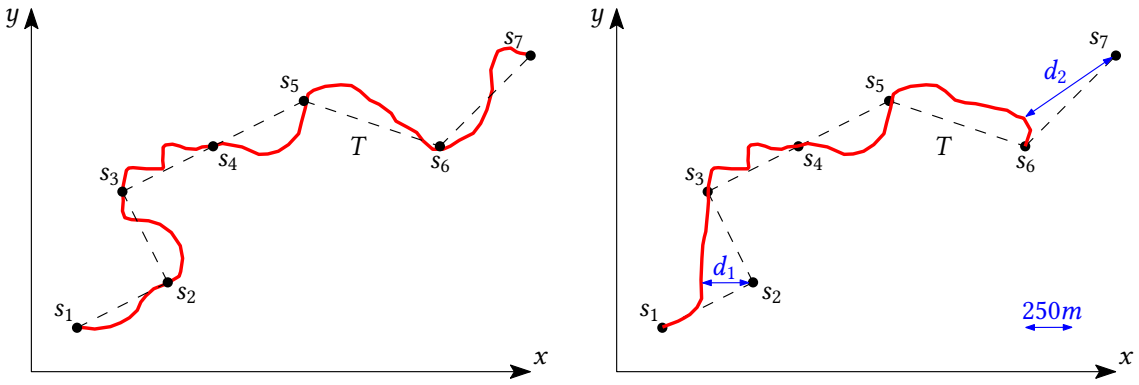


Figure 2.3: Left: A valid shape (red) for a trip T visiting stations $s1, s2, s3, s4, s5, s6, s7$, in that order. The dashed black line is the as-the-crow-flies shape we consider degenerate. Right: an erroneous shape (red) for the same trip. The distance d_1 between station s_1 and the shape is greater than the 250 meters threshold. The same holds for distance d_2.

For each year beginning with 2015, we downloaded all GTFS feeds that were active on March 1st of that year. We chose March 1st because many agencies publish new major versions of their timetables at the beginning of a new year. We assumed that this transition phase is over in spring. For each vehicle trip in this data, we evaluated the quality of the associated shape. We assigned the following categories to each trip:

NO-SHP The vehicle has no associated shape information.

ERR One or more stops of the vehicle trip had geographical coordinates with a distance greater than 250 meters to the associated shape. This error may occur in the following situations: (1) The assigned shape is the shape for another trip. (2) The trip has been updated in the past to serve different stops, but the shape was not updated accordingly. (3) The trip was assigned a "default" shape for a line, but has a slightly different stop sequence than the majority of trips traveling on the line (e.g. an outlier trip during rush hour, or because the trip is traveling back to some depot).

DEG The associated shape was degenerated. We say a shape is degenerated if it has exactly one anchor point for each served station. In other words, the shape is exactly the default as-the-crow-flies representation. In rare cases, the correct geographical path between each subsequent stop pair in a trip's stop sequence may be a straight line, which would then be falsely classified as degenerate.

OK None of the above apply, the shape is considered to be of good quality.

The results can be seen in Table 2.1. Both the number of feeds and the total number of trips contained in all of them have more than tripled since 2015. However, the percentage of trips that were provided with shape information *decreased* over the years investigated, from 60.4% in 2015 to 58.8 in 2020, further establishing the need for an

Table 2.1: Dimensions and shape quality of GTFS feeds published on `https://transitfeeds.com` between 2015 and 2020. Under "Feeds", we give the total number of feeds contained (TOT) and the number of feeds with shapes (HAS-SHPS). Under "Trips", we give the number of trips contained in all feeds (TOT), the number of trips with no shape (NO-SHP), the number of trips with erroneous shapes (ERR), the number of trips with degenerated shapes (DEG) and the number of trips with good shapes (OK).

	Feeds		Trips				
	TOT	HAS-SHPS	TOT	NO-SHP	ERR	DEG	OK
2015	292	258	5.1M	1.9M (39.1%)	9.5k (0.2%)	22.9k (0.5%)	3.1M (**60.4%**)
2016	440	399	6.4M	2.2M (34.7%)	20.7k (0.3%)	26.9k (0.4%)	4.1M (**64.6%**)
2017	642	573	11.2M	4.1M (36.5%)	50.0k (0.5%)	49.1k (0.4%)	7.1M (**62.6%**)
2018	667	605	11.7M	4.0M (34.3%)	53.5k (0.5%)	45.3k (0.4%)	7.6M (**64.9%**)
2019	708	653	11.9M	4.1M (34.1%)	33.5k (0.3%)	94.3k (0.8%)	7.7M (**64.8%**)
2020	906	805	17.8M	7.3M (40.4%)	88.4k (0.5%)	61.3k (0.3%)	10.5M (**58.8%**)

automated solution to generate these shapes. The setup for this survey can be found online[4].

We note that a particularly big schedule dataset is missing from transitfeeds.com: the entire schedule data of Germany. This dataset has recently been published in the NeTEx format and converted to GTFS. It contains over 2.3 million trips (more than 10 % of the number of trips contained in `transitfeeds.com`) and lacks any shape data. Based on repeated contact we had with public transportation authorities in Germany, it is our belief that such a nationwide schedule dataset with shape information does indeed not exist. This would make the dataset we generated in our experimental evaluation the first of its kind for an entire nation the size of Germany.

2.2 Preliminaries

This section introduces some additional concepts and definitions used in this chapter.

2.2.1 Probability Theory

Definition 2.1 (Events and Sample Spaces)**.** A sample space is a set Ω of outcomes $s_1, s_2, \ldots, s_n$ of an experiment. An event $E \subseteq \Omega$ is a subset of these outcomes. For example, the sample space of rolling a die is $\{1, 2, 3, 4, 5, 6\}$, and $A = \{2, 4, 6\}$ is the event of throwing an even number.

[4] `https://github.com/ad-freiburg/gtfs-shape-quality-survey`

Definition 2.2 (Probability Distribution)**.** A probability distribution describes the probabilities of events in a sample space. $\Pr(X = x)$ is the probability that the random variable X takes on value x. If X is discrete and can only take values $x_1, \ldots, x_n$, the probability distribution is given as a non-negative probability mass function $p(x) = \Pr(X = x)$ with $\Sigma_{i=1}^{n} p(x_i) = 1$. If X is continuous, its distribution is given as a probability density function.

Definition 2.3 (Probability Density Function)**.** The density of a continuous random variable X is a non-negative function $f(x)$ for which holds that (1) $\int_{-\infty}^{\infty} f_X(x)\,dx = 1$ and (2) $\int_a^b f_X(x)\,dx = \Pr(a \leq X < b)$.

Definition 2.4 (Cumulative Distribution Function)**.** The cumulative distribution function of a continuous random variable X is the function $F_X(x) = \Pr(X \leq x) = \int_{-\infty}^{x} f_X(y)\,dy$ that gives the probability that X is smaller than or equal to x. In particular, $1 - F_X(x) = \Pr(X > x)$. As $\Pr(X = x) = 0$ for any continuous random variable X and any x, $\Pr(X \leq x) = \Pr(X < x)$ and $\Pr(X \geq x) = \Pr(X > x)$.

Definition 2.5 (Conditional Probabilities)**.** Given two events A and B, the conditional probability $\Pr(A \mid B)$ is the probability that A occurs given that B already occurred.

2.2.2 Hidden Markov Model

As mentioned above, our approach is based on a hidden Markov model (HMM). It may be defined as follows:

Definition 2.6 (Hidden Markov Model)**.** A hidden Markov model (HMM) is a tuple (H, O, T, E, q_0) where $H = \{h_1, \ldots, h_n\}$ is the set of hidden states, $O = \{o_1, \ldots, o_m\}$ is the set of observations, $T = \{a_{ij}\}$, $a_{ij} = \Pr(h_j \mid h_i)$ gives the state transition probability distributions (the probability from going from state h_i to state h_j), $E = \{b_{ij}\}$, $b_{ij} = \Pr(o_j \mid h_i)$ gives the emission probability distributions (the probability that state h_i *emits* observation o_j). $q_0 \in S$ is an initial state.

The initial state is sometimes omitted and an initial state probability distribution is given instead [124]. If an initial state q_0 is defined, the transition probabilities from q_0 to any other state can be understood as this initial state distribution. One may now ask: given an *observation sequence* $S = s_1, s_2, \ldots, s_N$, where $s_k \in O$, what is the most likely sequence of hidden states $i_1, i_2, \ldots, i_N$, $i_k \in H$? Formally, we want to find

$$\underset{i_1,\ldots,i_N}{\arg\max} \prod_{k=1}^{N} \Pr(i_k \mid i_{k-1}) \cdot \Pr(s_k \mid i_k). \tag{2.1}$$

An often used $O(|O| \cdot |H|^2)$ algorithm for this problem is the Viterbi algorithm [140].

2.3 Related Work

Our work is closely related to previous work on map-matching GPS traces and schedule data, routing with turn restrictions, curve similarity measures (used in our experimental evaluation), string label similarity, and similarity measures for geographic locations.

2.3.1 Map-Matching of GPS Traces

The goal of map-matching GPS traces is to find a sequence of edges in a transportation network (usually a road network) that best matches a sequence $s_1, \ldots, s_n$ of sample points from a GPS device. A sample s_k may simply be the location measurement reported by the GPS device, but may also contain measurements regarding the current vehicle speed, orientation, elevation, or timestamps.

Map-matching of GPS traces is often divided into two categories: In *offline map-matching*, the sequence of sample points $s_1, \ldots, s_n$ is completely known at the time of map-matching. This is for example the case if measurements were taken during a trip and map-matched after the trip was completed. In this scenario, a major advantage is the knowledge of *driver intent* - it is known from the sample point sequence where the vehicle was headed, and this information may be included in the map-matching approach.

In *online map-matching*, the sequence of sample points $s_1, \ldots, s_n$ is *not* completely available at the time of map-matching, but only as a prefix up to a certain s_i. New measurements are continuously added to the sequence prefix. A typical example is a navigation device in a traveling car, where the current vehicle position and the route traveled so far are displayed on a map. Besides the lack of information about future sample points, the major challenge in online map-matching is the computation time.

Because of the high relevance of this problem for navigation systems, a vast amount of approaches has been described in the literature so far. For a recent survey, see for example [37]. [145] categorized map-matching approaches into three broad classes: (1) incremental max-weight, (2) global max-weight, and (3) global geometric. Both incremental and global max-weight methods consist of three steps: (1) for each sample point s_i, a set of candidate locations (e.g. road segments or graph nodes) is determined. (2) A weight is calculated for each of these candidate locations (e.g. the distance to the sample point, or a weight based on a comparison of the current candidate location to a previously chosen one). (3) The candidate sequence with the maximum weight is determined. Incremental max-weight methods greedily optimize the candidate sequence, and global max-weight methods find the globally optimal candidate sequence.

If max-weight based methods model the weights as probabilities and aggregate them by multiplication (essentially calculating the candidate sequence of highest probability) it is often called a *probabilistic* model. Typically, a probability distribution is defined

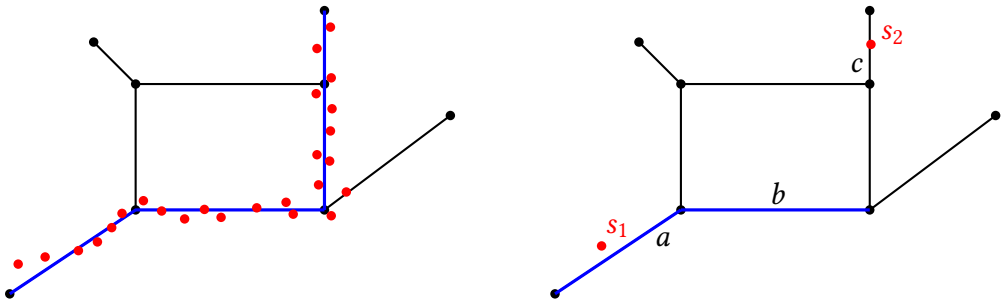


Figure 2.4: A typical problem of HMM based methods which only define non-zero transition probabilities to adjacent road segments. The red dots are sample points, the black lines are road segments, the black nodes are road intersections, and the most likely path through the network is depicted in blue. With a sufficiently high sampling rate, the correct path through the network is found (left). When confronted with very low sampling rates (right), such methods are typically not able to interpolate between unadjacent road segments. Road segment b is chosen as the candidate segment for sample point s_2, although c clearly would've been the better choice - but the transition probability from a to c is zero.

which gives the probability that a location candidate in the road network is *correct* for some measurement s_i. This is usually combined with a probability distribution which gives the probability that a location candidate is the *successor* of a previous candidate.

Regarding probabilistic models, approaches based on a hidden Markov model (HMM) have received considerable research attention in the past. Each candidate location is then a hidden state h in the HMM according to Definition 2.6, and each sample point s is an observation. The emission probabilities model how likely it is for a candidate location to be the correct choice for a sample point. The transition probabilities model how likely it is for a candidate location h to be the successor of a candidate location h'.

In [86] and [121], a map-matching approach based on a HMM was described which uses road segments as position candidates. Sample points consisted of GPS location measurements and vehicle orientations. The emission probabilities were modeled on (1) the angular difference between the vehicle orientation and the road segment orientation and (2) on the Euclidean distance between the GPS location and the road segment. Those distances (essentially the measurement errors) were assumed to follow a zero-mean Gaussian. Transition probabilities were modeled uniformly, with special penalties for transitions describing a U-turn. Non-zero transition probabilities between road segment candidates were only given to segments which shared a common node. This imposes a subtle, but important restriction: namely, that each road segment candidate must be "selected" by *at least one* sample point s to appear in the final map-matched path. It is not possible to "jump" over multiple road segments. Such a segment interpolation, however, is a common requirement for sparse sample points.

In [109] and independently in [104], this problem was identified and the idea of using shortest path calculations to both determine the transition probabilities and to interpolate between road segment candidates was developed. In both works, the emission probabilities were modeled only on the distance between the road segment candidate and the sample point, again assuming a zero-mean Gaussian distribution of the GPS noise. The transition probabilities were determined as follows: for each candidate h_j, consisting of a road segment r_j and a position p_j on this road segment, a shortest path query was run between p_j and the position $p_{j'}$ on the next candidate $h_{j'}$. The transition probability was then modeled in [109] based on the difference between the shortest path distance between p_j and $p_{j'}$ in meters and the as-the-crow-flies distance between p_j and $p_{j'}$ in meters. In [104], the ratio of these distances was chosen. Additionally, [104] based the transition probabilities on the similarity of the speed pattern between the sample points and the speed limits of the road segments. This may be error-prone, in particular because vehicle speed patterns may differ greatly from the speed limits of the used road, e.g. because of heavy traffic. With a sampling rate of 120 seconds, [109] report a ratio between the length of incorrectly matched segments and the total length of the correct route of around 0.16.

Eisner et al. [52] used a global max-weight approach which optimizes the length of the path connecting candidate sets for the sample point (selected using a threshold distance). To find the optimal path, a layered graph resembling the state graph found in HMM based approaches was used. A speedup technique was described which allows the calculation of all *required* shortest paths between two candidate sets with a single run of Dijkstra's algorithm. In Section 2.9.1, we extend this technique to also work in the HMM scenario, where emission probabilities must also be considered.

In [132], an approach was introduced which generates k path candidates by first calculating a shortest path from s_1 to s_n and then running $k-1$ additional shortest path searches from s_1 to s_n, where in each new search, the longest road-segment used in the previously found shortest path is added to a set of blocked road-segments. At the end, the path candidate with the smallest summed distance to all sample points $s_1, \ldots, s_{n-1}$ is chosen. Such an approach is obviously restricted to cases where the map-matched vehicle was following a shortest path from s_1 to s_n. This may be true in a goal-directed private car scenario (or a taxicab scenario, as analyzed by the authors), but not for a public transit vehicle, where the general priority is to serve as many people as possible en-route, not to find the shortest path from the first to the last station.

Some works also try to improve the map-matching quality under low sampling rates by leveraging additional meta-information. For example, in [119], a learning-based approach was used to analyze the number of turns relative to the travel distance tolerated by a human, which was then used during the map-matching. Zheng et al. [156] analyze historical trajectory data to derive travel patterns and use this data to suggest likely paths. Aly and Youssef [8] describe a HMM based approach that incorporates *road se-*

mantics detected by smartphone data (e.g. sharp turns, tunnels, bumpy roads).

A recent max-weight approach considers the case of *semi-restricted* trajectories [24]. These are trajectories for which the assumption that they follow the underlying transportation network is only partially correct, which may for example be the case if pedestrians cross large squares, or hike cross-country.

As the name tells, the max-weight methods described so far use weight parameters, which may require significant tuning. In contrast, *global geometric* approaches only use geometric similarities between the underlying network and the GPS trace to find the most likely path. Early approaches used the network path with the minimum Fréchet distance [63, 5, 30] by extending a method from [6] to graphs. In [30], the weak Fréchet distance was used instead, producing exactly the same paths in experiments (however, the weak Fréchet distance uses a non-monotonic parametrization and is hence not able to consider edge directions). This approach was further improved in [146].

Our method is also based on a hidden Markov model, and is therefore a global max-weight approach. The main differences between our work and existing work are: (1) The notion of what is considered sparse input data. In existing work, sampling rates of over 30 seconds [109], with an average distance between sample points of around 400 meters (assuming a vehicle travels with an average speed of 50 km/h) were already considered sparse. In [156], which specifically targeted sparsely sampled GPS traced, a sampling rate of over 2 minutes was considered sparse. In comparison, we have to handle cases where the average distance between sample points may be up to several hundred kilometers (for example, for long distance trains). (2) The method of transportation of the underlying network. Previous work on map-matching was mostly limited to road networks. In a public transit scenario, we must be able to do map matching also on rail or ferry networks. Here, physical limitations (for example, regarding acute turn angles) inherent to both the network and the vehicle type must be considered to achieve reasonable matching results. (3) The notion of what can be considered as *candidates* in the network graph for the placement of sample points. In previous work, candidates for GPS measurements were usually road segments or nodes in the graph modeling the network. In our case, candidate sets are limited to (or should prefer) *station nodes* in the underlying network.

2.3.2 Map-Matching of Schedule Data

There is only little work on map-matching of schedule data. In [31], the authors describe a greedy approach based on iterative shortest path calculations that is similar to our first baseline approach described in Section 2.4, but adds special auxiliary edges between node candidates to avoid stalling. A global approach based on the construction of a "pseudo-graph" was described in [122] and evaluated against the schedule data of Zurich on an OpenStreetMap (OSM) network. The punishment of full turns and turn restrictions

present in the OSM data as well as the usage of OSM attributes for better node candidate selection where left as an open problem. In contrast, our work considers both types. A preliminary version of our approach was published in [14].

2.3.3 Routing with Turn Restrictions

Our approach to handle turn restrictions is based on previous work in [149] and [71]. In [149], the concept of modeling turn restrictions by an edge-to-node dual graph was studied. In [71], the goal was to improve both the time and space efficiency of turn-restricted shortest path searches in road networks. An edge-based Dijkstra's algorithm was described that is similar to our routing approach on the edge-to-node dual graph (Section 2.6.4), and its applicability to Contraction Hierarchies was shown.

2.3.4 Curve Similarity

For our evaluation, we build on metrics for curve similarity and map-matching quality previously described in the literature. A discrete variant of the Fréchet distance was first described in [53] together with an $O(pq)$ algorithm for its computation, where p and q are the number of discrete sample points in the compared curves P and Q. A similarity metric between curves based on the average Fréchet distance is for example described in [32] and was first used as a quality metric for map-matching results in [30]. In [104], two additional metrics A_N and A_L based on the number and length of incorrectly matched segments were introduced, of which we use A_N in Section 2.10.

2.3.5 Similarity Classification of String Labels and POIs

The similarity measures used in our station similarity classification are related to previous work on string label similarity classification [29, 41, 78], often used for name disambiguation [57, 87]. Another area of research aims to find pairs of points of interest (POIs) describing the same real-world location [128, 129, 102, 46], which is for example required to merge different geographical datasets without overlap.

2.4 Baseline Approaches

At first glance, Problem 1 may seem like a classic routing problem: given an ordered list of destinations, find the optimal route which visits each destination in the given order. This apparent similarity gives rise to several baseline approaches, which we will shortly discuss in this section. We will gradually improve upon the simplest of these approaches until we arrive at a basic hidden Markov model formulation in the next section.

Given a vehicle trip T of type λ which visits stations $s_1, ..., s_n$ and a public transportation network $\mathcal{N}_\lambda = (V, E)$ with stations $\mathcal{S} \subseteq V$. Probably the simplest idea is to first select n candidate nodes $v_1, ..., v_n, v_i \in \mathcal{S}$ such that v_i is the candidate node for s_i, for example by taking the v_i closest to s_i. Afterwards, search for the shortest paths between each consecutive node candidates, with edge weights based on the travel time.

As there may be many fitting station nodes for a single s_i in $\mathcal{S}$, an obvious extension is to use candidate *sets* $S_i \subseteq \mathcal{S}$ for each station s_i. To determine the path between an S_i and its successors S_{i+1}, we then calculate the set-to-set shortest path between S_i and S_{i+1}. As described in Section 1.3.3, such a path can be found with a single run of Dijkstra's algorithm. However, when calculated separately, the shortest paths between individual pairs S_i and S_{i+1} are not necessarily connected. We see three ways to tackle this problem: node-to-set routing (NTS), local set-to-set routing (STS), and global set-to-set routing (G-STS). Figure 2.5 gives examples for all three.

2.4.1 Node-To-Set Routing

Here, the paths between the candidate sets are not calculated individually. Instead, we first find the shortest path from S_1 to S_2. This settles a node $v_2 \in S_2$, namely the target node of the found shortest path. For the path from S_2, we then only search for the optimal path from v_2 to S_3. The resulting node v_3 is then again settled as the node for S_3, and so on.

2.4.2 Local Set-to-Set Routing

If instead the shortest paths between station candidate set pairs are calculated individually, the resulting "gaps" between the individuals shortest path may be closed either by using a straight line segment (ignoring the network restrictions), or by calculating an additional shortest path between the target node $v_i \in S_i$ belonging to the path from S_{i-1}, and the source node $v_i' \in S_i$ belonging to the path to S_{i+1}.

2.4.3 Global Set-to-Set Routing

This variant searches for the globally optimal path p through $\mathcal{N}$ that visits *one* node of each candidate set $S_1, ..., S_n$ in the given order. Such a path can be found by first calculating the shortest path between each pair $(v, v') \in S_i \times S_{i+1}$. Let P_i denote the set of paths for pairs $S_i \times S_{i+1}$, and w.l.o.g. assume that these paths are given as node paths. Formally, we then search for a sequence $p^1, ..., p^{n-1}, p^i \in P_i$ which minimizes

$$\underset{p^1,...,p^{n-1}}{\arg\min} \sum_{p^i} c(p_i) \tag{2.2}$$

under the constraint that the last node in p^i is equal to the first node in p^{i+1}. This approach was used in [52]. The optimal path sequence can be found by building a directed acyclic graph (DAG) in which the S_i form node layers, it is hence often called a *layered graph*. For each candidate set S_i, all $v \in S_i$ are distinct nodes in this layer graph, and each node for a $v \in S_i$ is connected with an edge to each node for a $v' \in S_{i+1}$, weighted by the cost of the shortest path from v to v', and labeled with a reference to the corresponding path. Additionally, a special source node is connected to each $v \in S_1$ with an edge of weight 0, and each $v \in S_n$ is connected to a special target node with an edge of weight 0. The shortest path on this DAG from the source to the target node is then the optimal sequence $p_1, \ldots, p_{n-1}$.

2.5 Hidden Markov Model

Our baseline approaches so far ignored whether a station node candidate is the right choice for a particular s_i. In this section, we extend the G-STS approach by giving both node candidates and the paths connecting them a probability score. Given a vehicle trip to be map-matched, we then search for the most likely path the trip has taken through the transportation network. This can be elegantly modeled using a hidden Markov model.

The hidden states H are the network's station nodes (that is, $H = \mathcal{S}$). In the remainder of this chapter, we will use the terms hidden states and station candidates interchangeably, denoted by h. The observations O are the stops $\Sigma = s_1, \ldots, s_N$ visited by a trip T in that order (that is, $O = \Sigma$).

2.5.1 Emissions

Previous work usually modeled the emission probability $\Pr(s \mid h)$ of observing s for a location candidate h on the great-circle distance between s and h and assumed that this distance followed a zero-mean normal distribution [86, 121, 109, 104]. The intuition behind this is that the observed locations are prone to GPS measurement errors. It has been acknowledged in previous work that GPS measurement errors do in fact *not* follow a normal distribution [109].

We consider it a significantly different problem to estimate the probability that a GPS measurement matches a road segment than to estimate the probability that the coordinate of a public transport vehicle stop matches a station in the transportation network. Stop locations in schedule data are usually recorded manually, and there may be many conflicting philosophies of where to place the stop. Possible approaches are for example the exact position on the tracks where the vehicle comes to a halt, the centroid of the station area, the centroid of the station building, the entrance of the station building, or the platform.

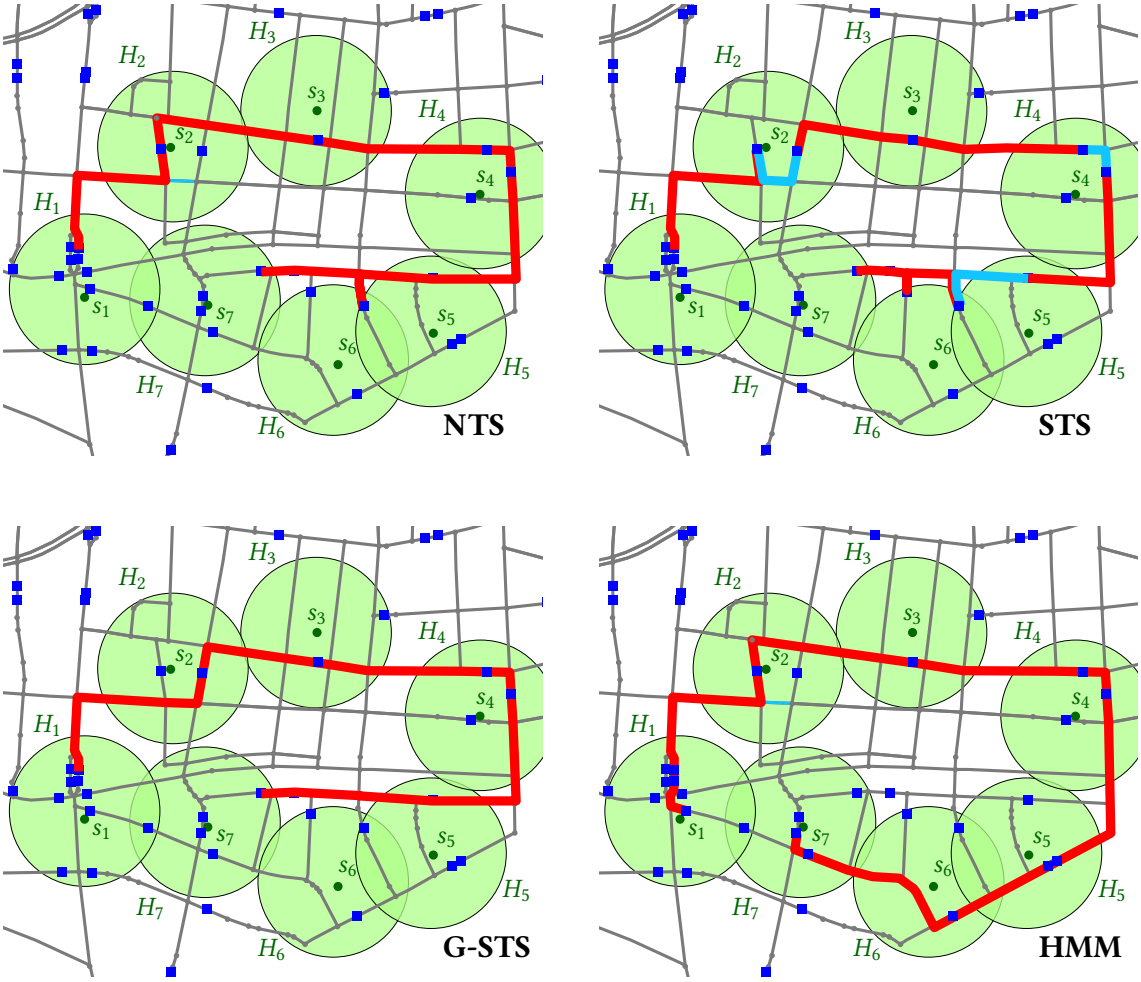


Figure 2.5: A transportation network $\mathcal{N}$ with a map-matching problem for a single vehicle trip T, solved using the node-to-set baseline (NTS), the set-to-set baseline (STS), the global set-to-set baseline (G-STS), and a hidden Markov model (HMM). The transportation network is depicted in black. The network's station nodes are depicted in red. The trip's station sequence $\Sigma = s_1, \ldots, s_7$ is depicted in dark green. All station nodes within the search radius (light green) around an s_i are used to build the set of station candidates H_i.

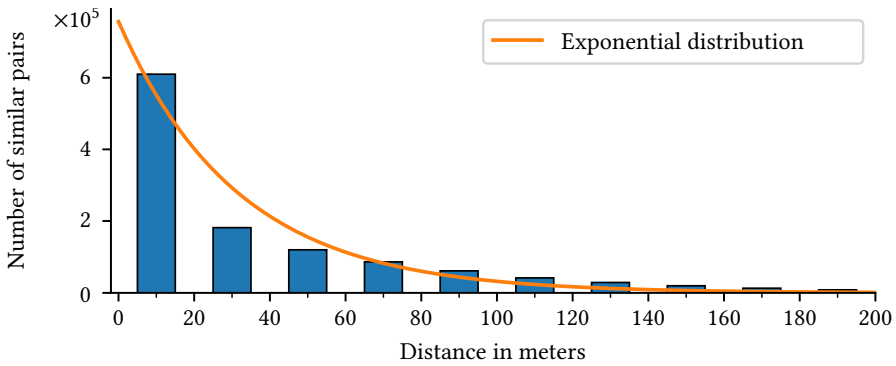


Figure 2.6: Distance distribution between similar station pairs in ground truth data obtained from OSM.

Based on an analysis [15] of stop positions belonging to the same station in ground truth data we obtained from OSM, it is reasonable to assume that the distance between these positions follows an exponential distribution (Figure 2.6). We therefore assume that the geographical distance between the position π of an observation s and the position $\Pi(h)$ of a node candidate h is a random variable D with the density function

$$p(d; \lambda_e) = \begin{cases} \lambda_e e^{-\lambda_e d} & \text{if } d \geq 0, \\ 0 & \text{else.} \end{cases} \tag{2.3}$$

For a hidden state h and an observation s with distance $d_{s,h}$, we then model the probability $\Pr(s \approx h)$ that h and s are a match as the probability that D is "big enough":

$$\Pr(s \approx h) = \Pr(D \geq d_{s,h}) = 1 - \Pr(D \leq d_{s,h}) \tag{2.4}$$

$$= 1 - \int_0^{d_{s,h}} \lambda_e e^{-\lambda_e d} \tag{2.5}$$

$$= e^{-\lambda_e d_{s,h}}. \tag{2.6}$$

We additionally normalize the $\Pr(s \approx h)$ to make sure that they sum up to 1. Let

$$C_e = \sum_{h \in H} \sum_{s \in \Sigma} \Pr(s \approx h). \tag{2.7}$$

We then model the emission probabilities as

$$\Pr(s \mid h) = \frac{1}{C_e} \Pr(s \approx h) \tag{2.8}$$

$$= \frac{1}{C_e} e^{-\lambda_e d_{s,h}}. \tag{2.9}$$

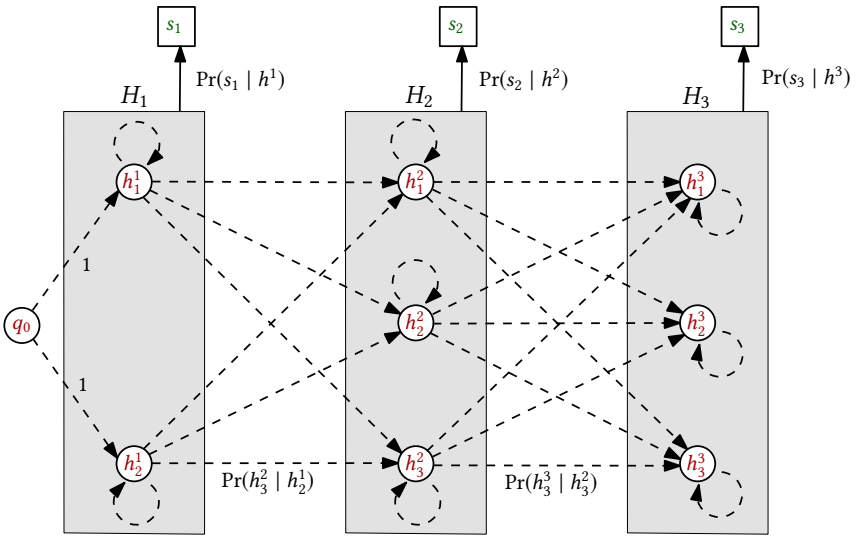


Figure 2.7: Hidden Markov model for a map-matching problem of a trip T with 3 stops s_1, s_2, and s_3. Dashed arrows denote transitions, solid arrows denote emissions. Transition and emissions with zero probabilities are omitted.

Observe that $C_e \geq \sum_{h \in H} e^{-\lambda_e d}$ for a single s and hence $\sum_{h \in H} \Pr(s \mid h) = \frac{1}{C_e} \sum_{h \in H} e^{-\lambda_e d} \leq 1$. For each h, the remaining probability weight $1 - \frac{1}{C_e} \sum_{h \in H} e^{-\lambda_e d}$ is put into an emission of a dummy *nil* observation $\varnothing$. We will see in the following section that this normalization factor has no influence on the optimal sequence of hidden states through the HMM.

For most pairs (s, h), $\Pr(s \mid h)$ will be very close to zero, as the geographical distance between them is very big. It is therefore reasonable to only calculate the emission probabilities for pairs that are geographically close (within a threshold distance r) and assume the remaining emission probabilities to be 0. For a given stop $s_i \in \Sigma$, we will use the following notation throughout the rest of this chapter: $H_i \subseteq H$ is the set of potential stop candidates within a distance r of s_i. We will sometimes denote by h^i that a stop node is a candidate for s_i, that is $h^i \in H_i$. In general, $\bigcup H_i \neq H$ (there might be network stations that are not within the threshold distance for any stop s) and $H_i \cap H_j \neq \emptyset$ for $i \neq j$ (stops might share the same candidate stations).

2.5.2 Transitions

We base the transition probability between two hidden states h and h' on the travel time t of the shortest path between them. Based on the intuition that a public transit vehicle will most likely travel to the next station candidate that is closest in terms of travel time, we model t as a random variable T following an exponential distribution. For a given t

and a small $\epsilon > 0$, the probability that T is within a range ϵ around t is then

$$\Pr(|T - t| < \epsilon) \approx \epsilon \lambda_t e^{-\lambda_t t}. \tag{2.10}$$

As with the emission probabilities in the previous section, we cannot use $\Pr(|T - t| < \epsilon)$ directly as the transition probability. To transform this continuous probability distribution over travel times into a discrete distribution over transitions from some hidden state h to a some hidden state h', we define

$$C_t = \sum_{h \in H} \sum_{h' \in H} \Pr(|T - t_{h,h'}| < \epsilon) \tag{2.11}$$

as the sum over all $\Pr(|T - t_{h,h'}| < \epsilon), (h, h') \in H \times H$, where $t_{h,h'}$ is the travel time of the shortest path between h and h'. We then model the transition probabilities as

$$\Pr(h' \mid h) = \frac{1}{C_t} \Pr(|T - t_{h,h'}| < \epsilon) \tag{2.12}$$

$$= \frac{1}{C'_t} e^{-\lambda_t t_{h,h'}}. \tag{2.13}$$

The remaining probability weight for each hidden state is put into a transition to itself.

The normalization by C_e and C'_t is strictly for conformity with the definition of a hidden Markov model and has no influence on the optimal sequence of hidden states, which will become clear in the following. For a given observation sequence $s_1, \ldots, s_N$, we now want to find the hidden state sequence $i_1, \ldots, i_N$ with maximum probability:

$$\arg\max_{i_1,\ldots,i_N} \Pr(i_1, \ldots, i_N \mid s_1, \ldots, s_N) \tag{2.14}$$

$$= \arg\max_{i_1,\ldots,i_N} \prod_{k=1}^{N} \Pr(i_k \mid i_{k-1}) \cdot \Pr(s_k \mid i_k) \tag{2.15}$$

$$= \arg\max_{i_1,\ldots,i_N} \frac{1}{C'_t C_e} \prod_{k=1}^{N} \Pr\left(\left|T - t_{i_{k-1},i_k}\right| < \epsilon\right) \cdot \Pr(s_k \approx i_k) \tag{2.16}$$

$$= \arg\max_{i_1,\ldots,i_N} \prod_{k=1}^{N} \Pr\left(\left|T - t_{i_{k-1},i_k}\right| < \epsilon\right) \cdot \Pr(s_k \approx i_k). \tag{2.17}$$

In the literature on map-matching using a HMM the transition and emission probabilities are typically given directly, without the normalization factor, although this does not strictly fit the HMM definition.

2.5.3 Transformation to Log Probabilities

As the probabilities tend to get very small, special care has to be taken to avoid numerical issues, in particular floating point underflow. A standard method is to optimize $\Pr(i_1, \ldots, i_N \mid s_1, \ldots, s_N)$ in log space:

$$\begin{aligned}
&\arg\max_{i_1,\ldots,i_N} \Pr(i_1, \ldots, i_N \mid s_1, \ldots, s_N) && (2.18)\\
=&\arg\max_{i_1,\ldots,i_N} \ln \Pr(i_1, \ldots, i_N \mid s_1, \ldots, s_N) && (2.19)\\
=&\arg\max_{i_1,\ldots,i_N} \ln \left(\prod_{k=1}^{N} \Pr\left(\left|T - t_{i_{k-1},i_k}\right| < \epsilon\right) \cdot \Pr(s_k \approx i_k) \right) && (2.20)\\
=&\arg\max_{i_1,\ldots,i_N} \sum_{k=1}^{N} \ln \Pr\left(\left|T - t_{i_{k-1},i_k}\right| < \epsilon\right) + \ln \Pr(s_k \approx i_k). && (2.21)
\end{aligned}$$

By dropping additive constants, we get

$$\begin{aligned}
&\arg\max_{i_1,\ldots,i_N} \Pr(i_1, \ldots, i_N \mid s_1, \ldots, s_N) && (2.22)\\
=&\arg\max_{i_1,\ldots,i_N} \sum_{k=1}^{N} -\lambda_t t_{i_{k-1},i_k} - \lambda_e d_{s_k,i_k} && (2.23)\\
=&\arg\min_{i_1,\ldots,i_N} \sum_{k=1}^{N} \lambda_t t_{i_{k-1},i_k} + \lambda_e d_{s_k,i_k}. && (2.24)
\end{aligned}$$

The scale factors λ_t and λ_e are parameters of our approach.

2.5.4 Optimal Hidden State Sequence as a Shortest Path Problem

From Equation 2.24 it immediately follows that we can find the optimal hidden state sequence $i_1, \ldots, i_N$ by calculating a shortest path in a special directed acyclic graph $U = (V_U, E_U)$. For each hidden state $h^i \in H_i$ we add a node to U. Additionally, we add a node for the initial state q_0 and a special sink node t. t is connected to each h^N by an edge with weight 0. q_0 is connected to each h^1 by an edge with weight $\lambda_e d_{s_1,h^1}$. The remaining nodes h^i with $1 < i < N$ are connected to each node h^{i+1} with an edge with weight $w(e) = \lambda_e d_{s_{i+1},h^{i+1}} + \lambda_t t_{h^i,h^{i+1}}$. Figure 2.8 gives an example.

It is easy to see that the node sequence of the shortest path from q_0 to t will then optimize Equation 2.24. As U is a directed acyclic graph, the shortest path can be found in $O(|V_U| + |E_U|)$ by first obtaining a topological ordering of the V_U, and then by processing them in that order, relaxing their adjacent neighbors. The key insight is that because of the topological ordering, if we process a node v, the cost label of v is already optimal

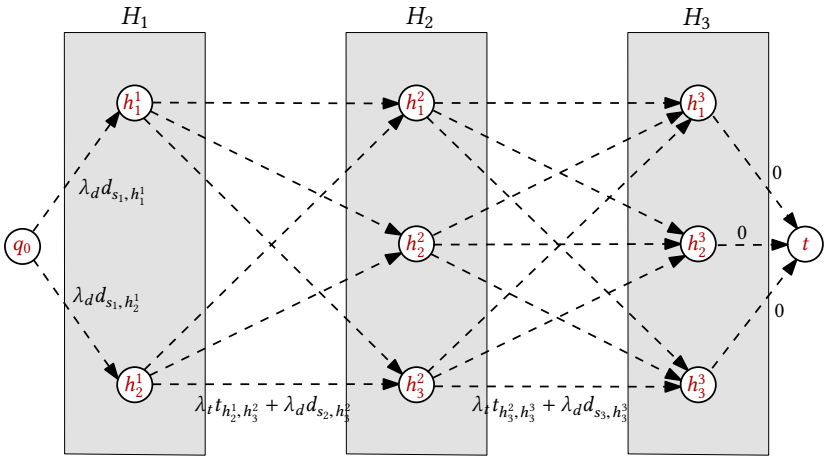


Figure 2.8: Finding the optimal hidden state sequence through the hidden Markov model from Figure 2.7 as a shortest path problem. Edge weights are depicted in black for 4 selected edges. The node sequence of the shortest path from source q_0 to sink t (without nodes q_0 and t) exactly corresponds to the optimal hidden state sequence $i_1, \ldots, i_N$.

(there cannot be a better path to v because we have already processed all possible predecessor nodes of v). Even better, because of the way we construct the HMM graph, we already have a topological ordering. Note that $|V_U| \in O(|H| \cdot |O|)$ and $|E_U| \in O(|O| \cdot |H|^2)$. The asymptotic running time can thus also be expressed as $O(|O| \cdot |H|^2)$, which is equivalent to the asymptotic running time of the Viterbi algorithm mentioned above.

2.6 Turn Costs

A standard issue in road network routing is handling turn restrictions. Consider Figure 2.9, left: a vehicle coming from some street e may not be allowed to enter into street f, whereas a vehicle coming from street h is allowed to enter. More generally, turning from one edge into another might not be restricted per se, but produce some cost:

Definition 2.7 (Turn Cost Function)**.** Given a directed graph $G = (V, E)$ and the set $A \subseteq E \times E$ of adjacent edges (we say (u, v) and (v, w) are adjacent, but not (u, v) and (w, v)). A turn cost function is a function $\mathcal{R} : A \mapsto \mathbb{R}_0^+$ that assigns two adjacent edges e and f a *turn cost* for traveling from e to f.

In the context of schedule map-matching, where sample points are typically very sparse, turn costs and turn restrictions have to be included into the shortest path calculations for the individual HMM transitions to ensure correct paths.

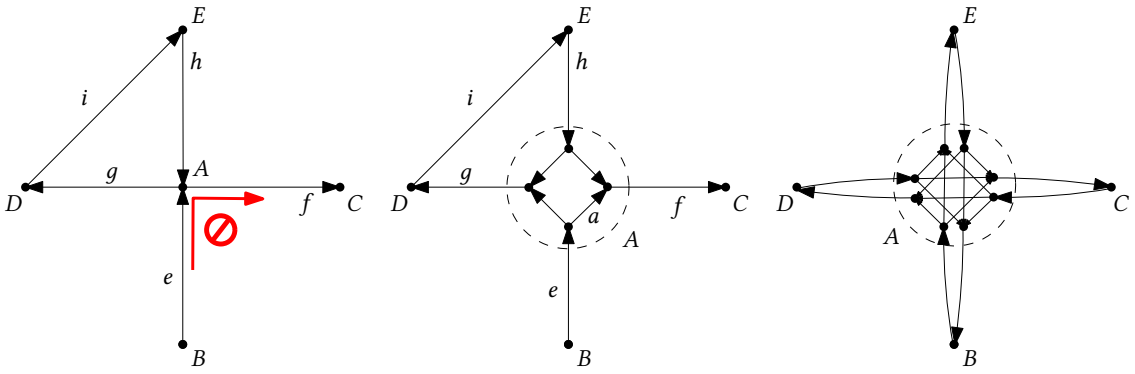


Figure 2.9: Left: coming from node B, there is a turn restriction at A that prohibits turning into segment f. Middle: a typical approach to model this is to blow up the node A to get explicit edges for each turn. In this example, the weight of edge a can now hold the turn cost. Right: if nodes are adjacent to many two-way street segments, the number of required edges quickly blows up

An often-used approach to model turn costs in directed graphs is shown in Figure 2.9, middle: affected nodes are expanded and "entry" nodes for each adjacent edge are added. Afterwards, edges for each possible turn are added, and may then be given weights corresponding to the turn costs. A drawback of this approach is that it may add $O(D)$ additional entry nodes, and $O(D^2)$ additional turn edges to the graph, where D is the maximum sum of the in- and out-degree for any node in the graph. For example, if n is a node in a street network with 4 adjacent roads which can each be used in both directions, we must expand n into 4 entry nodes, and add 3 edges to each entry node to cover all possible turns. Figure 2.9, right, gives an example. A first idea to slightly mitigate this is to contract the non-turn edges (for example, g, h, e, and f in Figure 2.9, middle). In this example, this leaves single nodes representing the original edges g, h, e, and f, namely D, E, B, and C. It is not immediately clear how to perform this contraction without losing information required for finding correct shortest paths later on, though.

This section first describes two basic types of turn costs that are relevant for schedule map-matching of road and rail bound public transit vehicles: mandatory turn restrictions and angular turn costs. We then continue to describe the problem of inter-hop turn costs (turn costs between hidden state transitions). The remainder of this section then extends and formalizes the edge contraction idea from the last paragraph and explains how turn costs can be elegantly modeled using the *edge-to-node dual* of the transportation network $\mathcal{N}$. In particular, we prove that no shortest paths are lost in this transformation and describe how to preserve all original edge weights in the process. We also show how inter-hop turn restriction can be naturally considered with this approach.

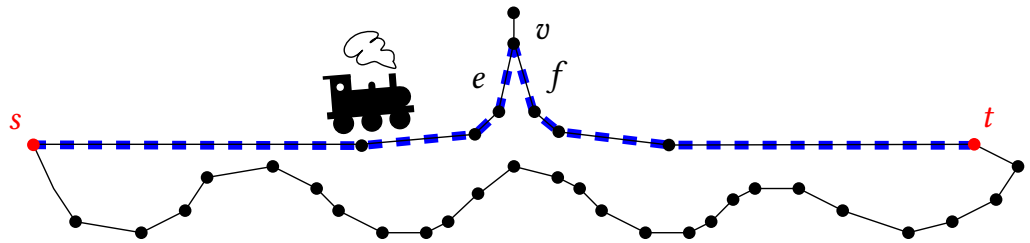


Figure 2.10: The shortest path from s to t is shown in blue - but a rail-bound vehicle is physically unable to make the sharp turn from edge e to edge f at node v. To make the lower path more likely, angular turn costs have to be considered.

2.6.1 Mandatory Turn Restrictions

Mandatory turn restrictions are regulated by traffic laws or (additionally) by physical barriers, for example a green corridor between road lanes. It is usually not possible to circumvent them, and it is therefore reasonable to model them as infinite turn costs.

Definition 2.8 (Turn Restrictor)**.** We call a turn cost function $\mathcal{R}^{\infty} \mapsto \{0, \infty\}$ a *turn restrictor.*

2.6.2 Angular Turn Costs

Depending on the vehicle type, we also have to consider turn restrictions stemming from physical limitations of the vehicle. Consider the example given in Figure 2.10. It is usually very time-consuming for a train to make a full turn on open track, although it would theoretically be possible (the train might stop, the driver could change to the rear cabin and continue in the opposite direction). Paths that use acute angles like the one in Figure 2.10 are immediately recognizable by humans as unrealistic. However, if we assume the edge weights in Figure 2.10 to be the travel time based on a uniform vehicle velocity, the path highlighted in blue is the optimal path from s to t. Similarly, it may take a bus some time to make an acute turn at an intersection, or to make a U-turn at a roundabout.

To consider this during map-matching, we add an angular turn cost function $\mathcal{R}^{\phi}$ which is directly based on the angle between two adjacent edges $e = (u, v)$ and $f = (v, w)$. As $\mathcal{N}$ is embedded in $\mathbb{R}^2$ by Π, this angle is given by $\phi(e, f) = \arccos(\hat{\mathbf{a}} \cdot \hat{\mathbf{b}})$, where $\mathbf{a} = \Pi(u) - \Pi(v)$ and $\mathbf{b} = \Pi(w) - \Pi(v)$.

2.6.3 Inter-Hop Turn Restrictions

During our map-matching process, we also want turn restrictions to be considered *between transitions.* We call these restrictions *inter-hop* turn restrictions. Consequentially,

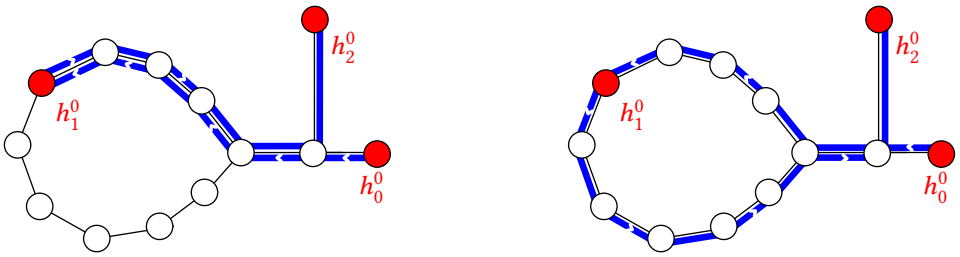


Figure 2.11: Shortest-path transitions between three hidden states h_0^0, h_1^0, and h_2^0. Left: turn restrictions are not considered between hops, leading to a full U turn if both shortest path transitions are combined. Right: turn restrictions are considered, leading to a smooth overall path.

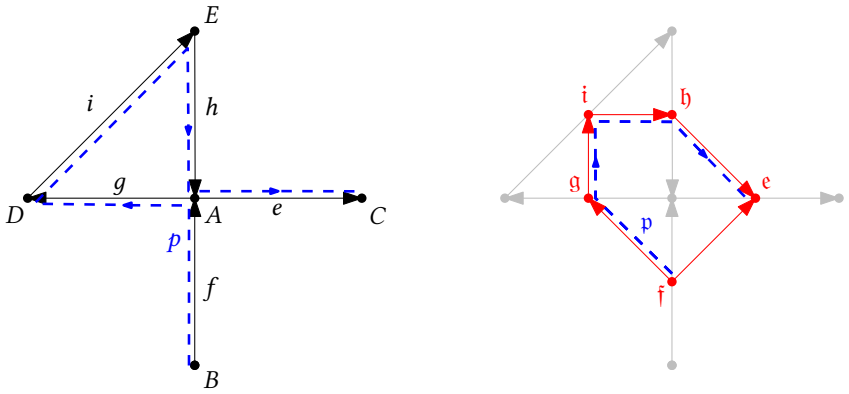


Figure 2.12: Left: A primal digraph G (black). Right: its edge-to-node dual $\mathfrak{D}(G)$ (red). The blue path $\mathfrak{p}$ from $\mathfrak{f}$ to $\mathfrak{e}$ is simple in $\mathfrak{D}(G)$, but its primal representation p from B to C in G is not (node A is visited two times).

turn restrictions that should be considered during a transition are sometimes called *intra-hop* turn restrictions. As the hidden states in our HMM are station nodes in the transportation network, and as a HMM is memoryless, it is not immediately clear how this can be ensured. Consider Figure 2.11, left. Two transition paths from hidden state h_0^0 to hidden state h_1^0, and from h_1^0 to h_2^0 are shown, as calculated by one shortest path search from h_0^0 to h_1^0, and one from h_1^0 to h_2^0. At the point of calculating the path from h_1^0 to h_2^0, we have "forgotten" the direction from which we arrived at h_1^0, and the calculated shortest path to h_2^0 will introduce a sharp U-turn at h_1^0. We would like our turn cost function to also take effect here. To achieve this, we have to encode the current vehicle heading into the hidden states. The basic idea, laid out in detail in the following section, is to not take nodes as hidden states, but edges.

2.6.4 Edge-to-Vertex Dual

The edge-to-node dual $\mathfrak{D}(G)$ of a graph G (then called the primal graph), first described in [147], is constructed by adding nodes for each edge of G, and by connecting two dual nodes if their primal edges were adjacent. Figure 2.12 gives an example. The edge-to-node dual is also often called *line graph*, but this term is already used for another concept in this work, so we use this more explicit name. Formally, the edge-to-node dual is defined as follows:

Definition 2.9 (Edge-to-Vertex Dual)**.** Given a primal graph $G = (V, E)$, we say $\mathfrak{D}(G) = (\mathfrak{V}, \mathfrak{E})$ is its *edge-to-node dual* if the following holds:

1. There is a bijection $\mathfrak{d}_\mathfrak{v} : E \mapsto \mathfrak{V}$ (each primal edge has a distinct dual node).
2. Let $\overline{E^2}$ be the set of adjacent edges in G (if G is directed, we say f is adjacent to e if head(e) = tail(f), otherwise we say f and g are adjacent if they share a node). There is a bijection $\mathfrak{d}_\mathfrak{e} : \overline{E^2} \mapsto \mathfrak{E}$ such that $\mathfrak{d}_\mathfrak{e}(e, f) = (\mathfrak{d}_\mathfrak{v}(e), \mathfrak{d}_\mathfrak{v}(f))$ (if two primal edges were adjacent, their dual nodes are adjacent).

We first prove several interesting properties which will help to clarify its applicability to the problem of turn restrictions.

Lemma 2.1. *For each simple primal path p with $|p| > 0$ through G there is a unique simple dual path $\mathfrak{p}$ through $\mathfrak{D}(G)$.*

Proof. As p is simple and $|p| > 0$ the path's edge sequence $e_1, \ldots, e_{n-1}$ implies a unique sequence of pairwise distinct nodes $\mathfrak{d}_v(e_1), \ldots, \mathfrak{d}_\mathfrak{v}(e_{n-1}) = \mathfrak{v}_1, \ldots, \mathfrak{v}_{n-1}$. Since each e_{i+1} is adjacent to e_i, $\mathfrak{v}_{i+1}$ is adjacent to $\mathfrak{v}_i$ by edge $\mathfrak{d}_\mathfrak{e}(e_i, e_{i+1})$. This dual node sequence therefore is a path $\mathfrak{p}$ through $\mathcal{D}(G)$. As the sequence was unique for p, so is $\mathfrak{p}$. □

We can therefore represent each nonempty simple path in G by exactly one distinct simple path in $\mathfrak{D}(G)$. There are no nonempty simple paths "lost" in the edge-to-node dual. An important consequence of this and Lemma 1.1 is that each nontrivial shortest path through G with edge weights > 0 has a unique dual representation.

Lemma 2.2. *For each simple dual path $\mathfrak{p}$ in $\mathfrak{D}(G)$ there is a unique primal path p.*

Proof. The path's node sequence $\mathfrak{v}_1, \ldots, \mathfrak{v}_n$ implies a unique edge sequence through G ${\mathfrak{d}_\mathfrak{v}}^{-1}(\mathfrak{v}_1), \ldots, \mathfrak{d}_v^{-1}(\mathfrak{v}_n)$. By Definition 2.9, this edge sequence is a path. □

Lemma 2.3. *This path p may not necessarily be simple.*

Proof. Consider the counter-example given in Figure 2.12. □

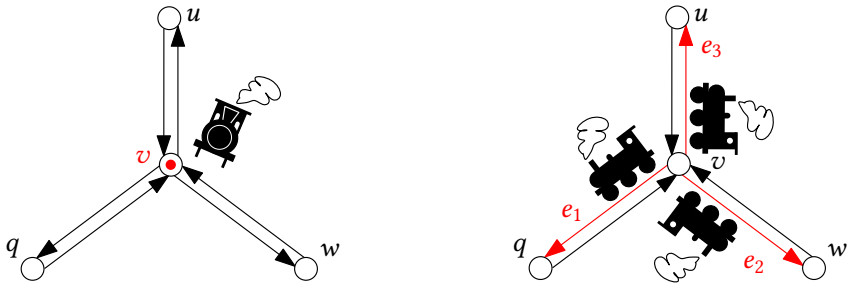


Figure 2.13: Left: the state of a vehicle being at node v is encoded by v itself, without any information regarding the direction it is headed to. Right: the state of a vehicle being at node v is encoded by all its outgoing edges. For example, e_3 encodes the state of being at v, heading in the direction of u.

Observe that $|p| = |\mathfrak{p}| + 1$ in general. To compute routes in the edge-to-node dual, we now search for shortest paths between nodes that correspond to primal edges and thus to segments in the underlying transportation network. Two questions immediately arise: (1) If we want to search for a shortest path between primal nodes s and t, what is a corresponding shortest path query in the edge-to-node dual? (2) How should the dual edge weights be modeled?

Candidate Edges

We now assume that the primal graph G is directed. As primal nodes have no immediate dual representation, we have to find a way to encode a primal shortest path query in the primal graph in a shortest path query in the edge-to-node dual. To this end, we represent a primal node v by a set $\Delta(v)$ of *gateway* edges. $\Delta(v)$ consists of all outgoing edges from v. For simplicity, we denote by $\mathfrak{d}_v(\Delta(v))$ the set of dual nodes corresponding to $\Delta(v)$.

Intuitively, an edge $e = (v, u) \in \Delta(v)$ and its corresponding dual node $\mathfrak{d}_v(e)$ encode the state of being at node v, heading towards u. Figure 2.13 gives an example.

To calculate the shortest path from primal nodes s to t in the edge-to-node dual, we now have to find the shortest path between the sets $\mathfrak{d}_v(\Delta(s))$ and $\mathfrak{d}_v(\Delta(t))$. This is again a standard set-to-set shortest path problem and can be computed by Dijkstra's algorithm as described in Section 1.3.3.

Dual Edge Weights

Special care has to be taken to ensure that primal edge costs are correctly reflected in the edge-to-node dual. In particular, if no turn costs are considered, we want the cost for a shortest primal path from s to t to be equivalent to the cost of the corresponding dual shortest path from $\mathfrak{d}_v(\Delta(s))$ and $\mathfrak{d}_v(\Delta(t))$.

Given a weighted graph G with a weight function w, we define the corresponding weight function in the edge-to-node dual as

$$\mathfrak{w}(\mathfrak{e}) = w\left(\mathfrak{d}_{\mathfrak{v}}^{-1}\left(\text{tail}\left(\mathfrak{e}\right)\right)\right). \tag{2.25}$$

In words: the weight of a dual edge $\mathfrak{e} = (\mathfrak{u}, \mathfrak{v})$ is the weight of the primal edge corresponding to its tail node.

Lemma 2.4. *Let $\mathfrak{p}$ be a path in $\mathfrak{D}(G)$ with edge sequence $\mathfrak{e}_1, \ldots, \mathfrak{e}_{|\mathfrak{p}|}$ and ending at node $\mathfrak{t}$. Its corresponding primal path is p with edge sequence $e_1, \ldots, e_{|\mathfrak{p}|+1}$. Let $\mathfrak{p}'$ the path $\mathfrak{p}$ extended by* any *node $\mathfrak{t}' \in \text{adj}^+(\mathfrak{t})$. Let G be weighted by w and let $\mathfrak{D}(G)$ be weighted according to Equation 2.25. Then $c(p) = \mathfrak{c}(\mathfrak{p}')$.*

Proof.

$$\mathfrak{c}(\mathfrak{p}') = \sum_{i=1}^{|\mathfrak{p}|+1} w\left(\mathfrak{d}_{\mathfrak{v}}^{-1}\left(\text{tail}\left(\mathfrak{e}_i\right)\right)\right) \tag{2.26}$$

$$= w(e_1) + \cdots + w(e_{|\mathfrak{p}|+1}) \tag{2.27}$$

$$= \sum_{i=1}^{|p|} w(e_i) = c(p). \tag{2.28}$$

□

Lemma 2.5. *Let G be a primal graph weighted by $w > 0$ and its edge-to-node dual $\mathfrak{D}(G)$ be weighted by $\mathfrak{w}$ according to Equation 2.25. The shortest dual path from $\mathfrak{d}_{\mathfrak{v}}(\Delta(s))$ to $\mathfrak{d}_{\mathfrak{v}}(\Delta(t))$ describes a shortest simple primal path from s to t.*

Proof. Let $\mathfrak{p}'$ be such a path with edge sequence $\mathfrak{e}_1, \ldots, \mathfrak{e}_{n+1}$ and cost $\mathfrak{c}(\mathfrak{p}')$. Observe that $\text{tail}(\mathfrak{d}_{\mathfrak{v}}^{-1}(\mathfrak{e}_1)) = s$ and $\text{tail}(\mathfrak{d}_{\mathfrak{v}}^{-1}(\mathfrak{e}_{n+1})) = \text{head}(\mathfrak{d}_{\mathfrak{v}}^{-1}(\mathfrak{e}_n)) = t$. Let p be the path p' with the last edge removed. According to Lemma 2.2, $\mathfrak{p}$ describes a unique path p in G, and this will start at s and end at t. It remains to show that p is really a shortest simple path from s to t. Assume that is not the case and that there is a primal path q from s to t with $c(q) < c(p)$. According to Lemma 2.1, this new path q also has a unique dual representation $\mathfrak{q}$. As q still begins at s, $\mathfrak{q}$ still begins at a node in $\mathfrak{d}_{\mathfrak{v}}(\Delta(s))$. As q still ends at t, $\mathfrak{q}$ still ends at a node adjacent to $\mathfrak{d}_{\mathfrak{v}}(\Delta(t))$. We can thus extend q by a single dual node into a path q' which ends in $\mathfrak{d}_{\mathfrak{v}}(\Delta(t))$. But according to Lemma 2.4, $\mathfrak{c}(\mathfrak{q}') = c(q)$, and therefore $\mathfrak{c}(\mathfrak{q}') < \mathfrak{c}(\mathfrak{p}')$, which means $\mathfrak{p}'$ was not the shortest dual path, a contradiction. p therefore was a shortest path from s to t. As $w > 0$, it was also simple. □

Figure 2.14 gives an example of a shortest path problem in a primal graph and the corresponding shortest path problem in the edge-to-node dual.

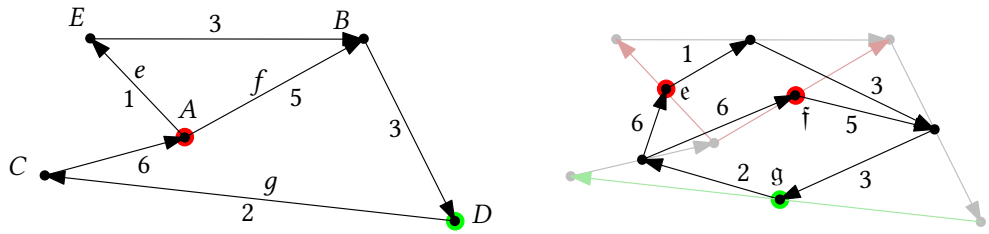


Figure 2.14: Left: a shortest path problem from A (red) to D (green) in a weighted primal graph G. Numerical labels depict the edge weights. Right: the corresponding shortest path problem in its edge-to-node dual. Instead of single source and target nodes, we now have a set of dual source nodes $\{\mathfrak{e}, \mathfrak{f}\}$ (red) corresponding to primal edges (light red) and a set of dual target nodes $\{\mathfrak{g}\}$ (green) corresponding to primal edges (light green).

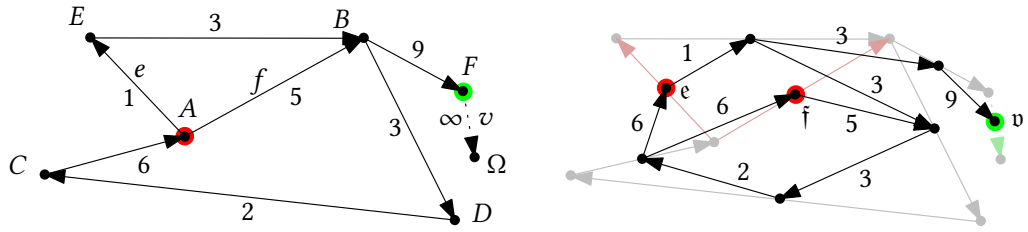


Figure 2.15: Left: a shortest path problem from A (red) to D (green) in a weighted primal graph G. Node F has no outgoing edge. To ensure a non-empty target node set in the edge-to-node dual, we add a virtual node Ω and a virtual edge $v = (F, \Omega)$. Right: the corresponding dual shortest path problem.

We have so far assumed that the sets $\Delta(s)$ and $\Delta(t)$ are nonempty, which may not be the case. For $\Delta(s)$, this is irrelevant, as we cannot find a shortest path from a node s to any other node if s has no outgoing edge. For t, however, a lack of outgoing primal edges means we cannot arrive at node t in the edge-to-node dual. Consider Figure 2.15. To be able to calculate a shortest path from A to F in the edge-to-node dual, we temporarily insert a virtual node Ω and a virtual edge $v = (F, \Omega)$. The weight $w(v)$ is set to infinity. As Ω will have no outgoing edge, $w(v)$ will not appear in any dual edge weight, but we are now able to calculate a shortest path to F (Figure 2.15, right).

Turn Costs in the Edge-To-Vertex Dual

In the edge-to-node dual, the consideration of angular, mandatory, and inter-hop turn costs is now straightforward. Given a turn cost function $\mathcal{R}$, we may simply update the dual weight function from Equation 2.25 to

$$\mathfrak{w}(\mathfrak{e}) = w\left(\mathfrak{d}_{\mathfrak{v}}^{-1}(\text{tail}(\mathfrak{e}))\right) + \mathcal{R}\left(\mathfrak{d}_{\mathfrak{v}}^{-1}(\text{head}(\mathfrak{e})), \mathfrak{d}_{\mathfrak{v}}^{-1}(\text{tail}(\mathfrak{e}))\right). \tag{2.29}$$

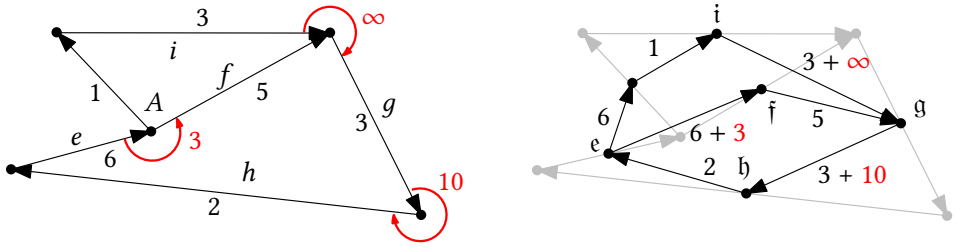


Figure 2.16: Left: a weighted graph G with a turn cost function $\mathcal{R}$. Non-zero turn costs between edges are depicted in red, for example $\mathcal{R}(e, f) = 3$. Right: the corresponding dual shortest path problem, turn costs are added to the corresponding dual edge.

See Figure 2.16 for an example.

To finally be able to consider inter-hop turn restrictions as explained above, we slightly change the definition of our hidden Markov model from Section 2.2.2. As hidden states H, we now no longer consider network station nodes $v \in \mathcal{S}$, but their outgoing edges: $H = \bigcup_{v \in \mathcal{S}} \Delta(v)$. As explained above, a hidden state now no longer encodes being orientationless at a station, but being at a station and heading in a particular direction.

A subtle point has to be addressed before we can close this section. As we are choosing the outgoing edges $\Delta(v)$ as new hidden states for a network station $v \in \mathcal{S}$, a turn restriction for entering an edge $e \in \Delta(v)$ might be added to the path cost already when we arrive at v, although the vehicle has not yet entered the network segment corresponding to e. If v is not the terminus for a map-matched trip, this is desired, as we have to leave through *some* outgoing edge anyway for the next hop, at which point this turn cost would actually apply. For a terminus station, however, this is not the case.

A simple solution to this problem is to add virtual nodes and edges like described above for all terminus station candidates, with no attached turn restrictions.

2.7 Incorporating Network Meta Data

Real-world datasets for public transportation networks often come with additional metadata that goes beyond the basic representation of network segments. This includes for example mandatory turn restrictions as already described, one-way streets, or speed limits. While the details of how such data can be transformed into a routable transportation network $\mathcal{N}$ would be out of scope here, there are two kinds of relevant metadata specific to our problem: public transportation routes, and station labels. This section will briefly describe how to incorporate them into our approach. We will focus on the largest dataset for transportation networks, OSM, which is also used in our experimental evaluation.

2.7.1 Public Transit Route Relations

OSM already contains information on public transportation routes via the relation `route`[5]. For example, relations of type `route=train` group network segments which belong to the route of a single train line. This line is specified via an attribute `line=*` which gives the line name (e.g. "Intercity-Express Basel SBB -> Hamburg Altona") and/or a `ref=*` attribute which gives the train number (e.g. "ICE 445"). Similar relations exist for other public transport vehicle types, for example streetcars, busses, or ferries.

At first glance, it may seem sufficient to extract a matching `route` relation for a vehicle trip we want to map-match. There are several reasons why this approach does not work: (1) The geographical coverage of `route` relations is not good enough, especially in rural areas. (2) The schedule coverage is usually not good enough. In particular, outlier trips (vehicles returning to the depot, additional irregular services during peak hours, etc.) are often missing. (3) The relations are often outdated. With each new schedule, `route` relations have to be updated manually. As these relations do not prominently appear in the standard OSM map tiles, outdated routes often remain unnoticed. (4) The relations itself are sometimes erroneous or have holes.

Despite these issues, `route` relations provide valuable information on the vehicle path. To incorporate this, we label each edge contained in a public transit `route` relation with the attribute value `line=*` and `ref=*`. Let now $L(e)$ be the set of line labels for an edge e in our transportation network, taken from the OSM attributes, and let l be the line label of a trip T. Instead of static edge weights $w(e)$ based on the traversal time we now use an edge weight function $f : E \times \mathcal{T} \mapsto \mathbb{R}^+$:

$$f(e, T) = \begin{cases} w(e) & \text{if } l \in L(e), \\ w(e) \cdot w_L & \text{else.} \end{cases} \tag{2.30}$$

Here, $w_L \geq 1$ is a penalty factor for edges with no matching lines.

2.7.2 Station Labels

Similarly, we would like to prefer network station candidates with a label that matches the station label in the schedule data. To this end, we add the station label attributes `name` and `uic_name` to the label set $N(v)$ of the corresponding network node v. To prefer network station node candidates with matching labels, we update our emission probabilities and now calculate the probability $\Pr(s \approx h)$ that a schedule stop s matches a network station node candidate h as follows:

[5] `https://wiki.openstreetmap.org/wiki/Public_transport#Service_routes`

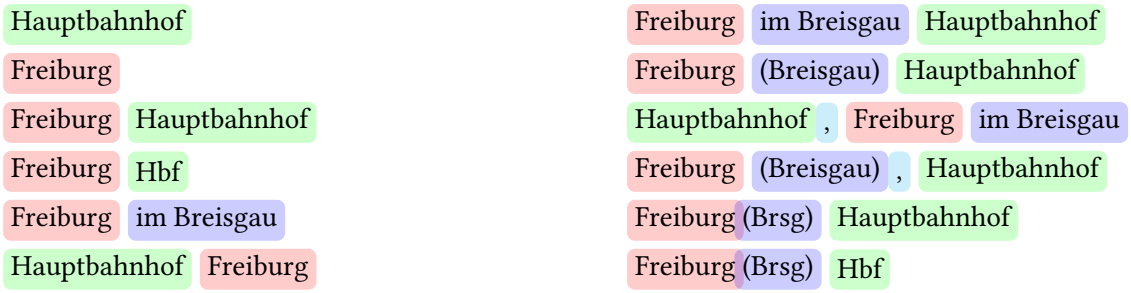


Figure 2.17: Some labeling variants for Freiburg Main Station. Same colors symbolize semantically equivalent tokens.

$$Pr(s \approx h) = \begin{cases} \Pr(D \geq d_{s,h}) & \text{if } n \in N(h), \\ \Pr(D \geq d_{s,h}) \cdot w_N & \text{else,} \end{cases} \tag{2.31}$$

where n is the label of scheduled stop s, and $w_N \geq 1$ is a penalty factor.

2.8 Advanced Station Similarity Classification

A problem with the updated emission probabilities from above is that stations are typically not labeled consistently in real-world datasets. A hard equivalency check would penalize a station node candidate labeled "London St Pancras" for a scheduled stop labeled "London, St Pancras", although they might both have the exact same geographical position. Clearly this is not desired. Instead, we would like to have a robust classifier to decide whether two stations are similar. We define such a *station identifier* z as a triple (n, ϕ, λ). n is the station name, ϕ its geographical latitude, and λ its geographical longitude. This section then studies the following problem:

Problem 2 (Station Similarity Classification)**.** *Find a classification function c that maps pairs (z_a, z_b) of station identifiers to $\{0, 1\}$. We want $c(z_a, z_b) = 1$ if z_a and z_b describe the same real-world station, and $c(z_a, z_b) = 0$ otherwise.*

This section describes the characteristics of station identifiers as they typically appear in real-world datasets. We then discuss several approaches to Problem 2. All approaches will be evaluated in Section 2.10.

2.8.1 Characteristics of Station Identifiers

Both the geographic coordinates and labels between similar station identifiers may differ significantly. Figure 2.17 gives an (incomplete) list of labeling variants for Freiburg Hauptbahnhof (the main station in Freiburg, Germany) as they appear in OSM and

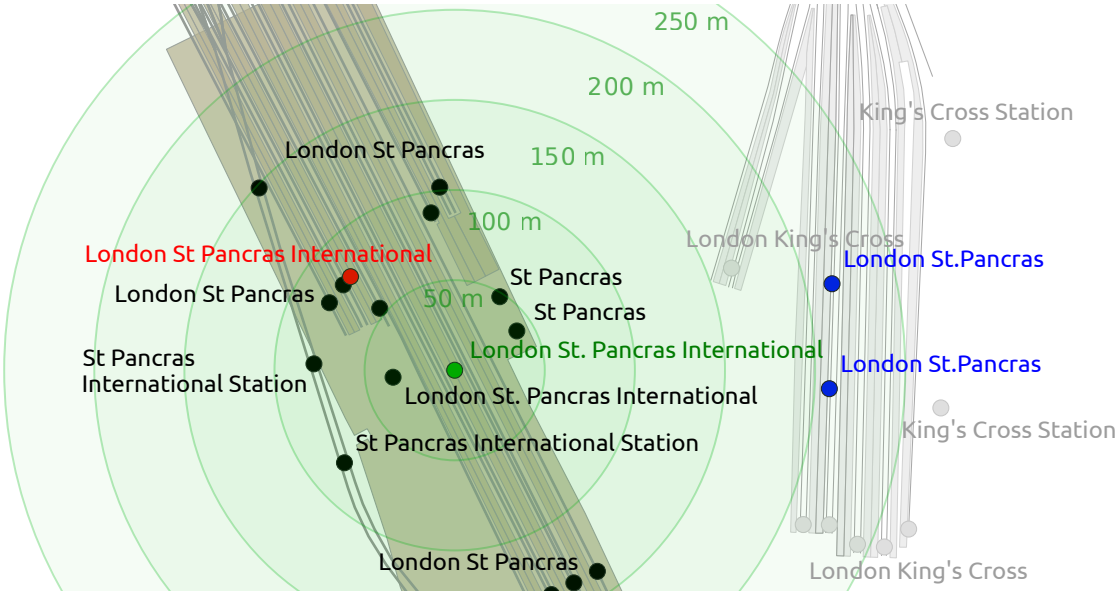


Figure 2.18: Station identifiers for London St Pancras in three different datasets: OpenStreetMap (● OSM, black), Deutsche Bahn schedule (● DB, green), Association of Train Operating Companies schedule (● ATOC, red), EuroStar schedule (● ES, blue). The ● DB station has a distance of over 200 meters to the ● ES stations. For both ● ES stations, the nearest station in the OSM dataset is King's Cross station.

schedule data for the local transportation authority. Similarly, Figure 2.18 shows several labeling variants for London St Pancras. In these examples, many of the typical characteristics of station identifier labels in western languages are present:

Few typos. Typos in station labels are rare. Station labels are very short and often displayed prominently, where typos would be quickly noticed.

Abbreviations. Station labels often contain abbreviations. These are either specific to a public transportation network, regionally specific (like "Freiburg (Brsg)" for "Freiburg im Breisgau"), typical for entire countries (like "Hbf" or "HB" for "Hauptbahnhof" in Germany and Switzerland, respectively) or language specific (like "Str." or "St." for "street" in German and English, respectively). Often the abbreviations are completely regional and hard to decipher even for locals (like "BBrunnen" for "Bertoldsbrunnen" in Freiburg).

Varying token separators. Tokens may be separated by space, varying punctuation (commas, semicolons, hyphens, brackets), or camel casing, among others.

Omitted location specifiers. Tokens which specify a geographic location (countries, cities, districts etc.) may be omitted either completely or partially, either because they are clear from context, or because the data is regional and just includes a single city.

Omitted station specifiers. Tokens which exactly specify a station may be omitted. This is often the case in long-distance schedule data, where trains only make a single stop per city, and where cities only have a single major railway station.

Different token ordering. Token ordering is often arbitrary. For example, "Freiburg Hauptbahnhof" and "Hauptbahnhof Freiburg" are equivalent.

There may also be considerable differences between the coordinates of similar station identifiers. Figure 2.18 depicts the position of several identifiers for London St Pancras station in OSM, the official Deutsche Bahn schedule, the official schedule of the Association of Train Operating Companies (ATOC), and the schedule data for the EuroStar lines. Again, several of the typical characteristics of station identifier coordinates are present:

Precision issues. Coordinates are often imprecise, e.g. because of truncation, or because they were manually compiled by clicking on a map.

Different placement philosophies. Station identifiers might be placed at the station entrance (there might be more than one), at the centroid of the station building (there might also be more than one building), at the centroid of the station polygon (the latter is not well-defined), at the centroid of the platform the vehicle arrives at, at the exact position the vehicle will come to a halt, at the exact position the center of the vehicle will *on average* come to a halt, or at a position that is best suited to render a station label.

Human error. The location of a station identifier might simply be wrong because of human error. For example, a station identifier coordinate might have been erroneously assigned to a neighboring station.

2.8.2 Classification Techniques

In order to tackle these characteristics, this section describers several techniques for station similarity classification. We first describe some naive baseline approaches which will later be also evaluated for comparison. Afterwards, some techniques for label similarity based on established measures for string similarity are discussed. We close with feature engineering techniques to make the problem accessible to machine learning.

Position Equivalency

A naive classifier would just check the positions of the station identifiers for equivalency, using some ϵ to account for floating point inaccuracies.

$$c_{PEQ}(z_a, z_b) = \begin{cases} 1 & \text{if } \mathrm{dist}(\phi_a, \lambda_a, \phi_b, \lambda_b) < \epsilon, \\ 0 & \text{else.} \end{cases} \tag{2.32}$$

It is trivial to construct examples where c_{PEQ} would produce false negatives.

Label Equivalency

In Section 2.7.2, we implicitly used a classifier based on label equivalency:

$$c_{LEQ}(z_a, z_b) = \begin{cases} 1 & \text{if } n_a = n_b, \\ 0 & \text{else.} \end{cases} \tag{2.33}$$

As described above, such a classifier would lead to many false negatives if used in a standalone fashion, but also to many false positives, as different stations in different cities might share the same label.

Label Normalization

Differences in spelling, common abbreviations, problems with lower- and uppercase, and redundant tokens (e.g. a station label suffixed with "station") might be normalized by manually created normalization rules.

Position Similarity

We now turn to classification based on label similarity measures. In the context of this work, a similarity measure is a function that maps to $[0, 1]$ and quantifies the similarity between its arguments (a higher value means higher similarity). To use these similarity measures for classification, a threshold value is defined.

The most basic similarity measure between station identifiers is based on the geographical distance. We recall Figure 2.6 and note that the geographical distance between matching station identifiers in real-world data seems to fit an exponential distribution. Correspondingly, we model the *position similarity* sim_{POS} as follows:

$$\text{sim}_{\text{POS}} = \exp\left(-\lambda \cdot \text{dist}\left(\phi_1, \lambda_1, \phi_2, \lambda_2\right)\right) \tag{2.34}$$

We would like to have $\text{sim}_{\text{POS}}(z_a, z_b) = 0.5$ if z_a and z_b have a distance $\hat{d}$ (the threshold distance) and therefore set $\lambda = (\ln 2)/\hat{d}$.

Edit Distance Similarity

A classic string metric is the edit distance, or Levenshtein distance. It gives the minimum number of edit operations (either insert, deletion or substitution) necessary to transform a string a into a string b. The edit distance is symmetric, i.e. $\text{ed}(a, b) = \text{ed}(b, a)$. An often used similarity measure based on the edit distance uses the ratio between $\text{ed}(a, b)$ and

the length of the longer string:

$$\mathrm{sim}_{\mathrm{ED}}(z_a, z_b) = 1 - \frac{\mathrm{ed}(n_a, n_b)}{\max(|n_a|, |n_b|)}. \tag{2.35}$$

It is again easy to construct examples where this measure underestimates similarity: with n_a = "Freiburg im Breisgau, Hauptbahnhof" and n_b = "Hauptbahnhof, Freiburg im Breisgau", $\mathrm{sim}_{\mathrm{ED}}(z_a, z_b) = 0.17$, although they clearly describe the same station.

Prefix and Suffix Edit Distance

With n_a = "St Pancras" and n_b = "St Pancras International Station", $\mathrm{sim}_{\mathrm{ED}}(z_a, z_b) = 0.31$. To get a similarity measure that is more robust against missing tokens, we could use the prefix or suffix edit distance. The prefix edit distance (PED) is defined as

$$\mathrm{ped}(a, b) = \min_{b'} \mathrm{ed}(a, b'). \tag{2.36}$$

where b' is a prefix of b. Similarly, the suffix edit distance SED may be defined. In this case, b' is then a suffix of b. Neither the PED nor the SED are symmetric. To still get a symmetric similarity measure, we compute both directions and take the best result:

$$\mathrm{sim}_{\mathrm{PED}}(z_a, z_b) = 1 - \min\left(\frac{\mathrm{ped}\,(n_a, n_b)}{|n_a|}, \frac{\mathrm{ped}\,(n_b, n_a)}{|n_b|}\right). \tag{2.37}$$

Then, again with n_a = "St Pancras" and n_b = "St Pancras International Station", $\mathrm{sim}_{\mathrm{PED}}(z_a, z_b) = \mathrm{sim}_{\mathrm{PED}}(z_b, z_a) = 1$. A similarity measure for the SED can be analogously defined. It is still easy to construct examples where this measure overestimates similarity. For example, the station labels "Berlin Zoo" and "Berlin Gesundbrunnen" have a PED similarity of 0.7, but describe two different railway stations in Berlin.

Jaro Similarity

The Jaro Similarity [89] is especially well suited for comparison of short strings. Given two strings a and b, it is defined as the average between the percentage of characters in a which have (withing a maximum distance of $\left\lfloor \frac{1}{2} \max(|a|, |b|) \right\rfloor - 1$) a matching character in b, the percentage of characters in b which have a matching character in a, and the percentage of matching characters which are exactly at the same position in a and b. The number of matching characters within the distance threshold is denoted by m, the number of matching characters which are not exactly at the same position is denoted by

t and often called the number of *transpositions*. The Jaro Similarity is then defined as

$$\text{sim}_\text{J}(z_a, z_b) = \frac{1}{3}\left(\frac{m}{|n_a|} + \frac{m}{|n_b|} + \frac{m-t}{m}\right). \tag{2.38}$$

Jaro-Winkler Similarity

To give strings which match from the beginning a slight advantage, Winkler [148] proposed to combine the Jaro Similarity $\text{sim}_\text{J}(z_a, z_b)$ with $1 - \text{sim}_\text{J}(z_a, z_b)$, weighted by the length $l < 5$ of the common prefix (we check only up to character 4 if the prefixes match) and a scale factor $p <= \frac{1}{4}$. It is then called the Jaro-Winkler Similarity.

$$\text{sim}_\text{JW}(z_a, z_b) = \text{sim}_\text{J}(z_a, z_b) + lp\left(1 - \text{sim}_\text{J}(z_a, z_b)\right). \tag{2.39}$$

Note that p cannot exceed $\frac{1}{4}$, because otherwise the similarity may be greater than 1. In practice, p is usually set to 0.1.

Jaccard Index

Another class of string similarity measures is based on tokens. While the PED and/or the SED similarity can account for missing tokens at the beginning or the end of station labels, it doesn't handle cases well where tokens are completely scrambled (or inverted). For example, for n_a ="Hauptbahnhof Freiburg" and n_b ="Freiburg Hauptbahnhof", both $\text{sim}_\text{PED}(z_a, z_b)$ and $\text{sim}_\text{PED}(z_a, z_b)$ are only 0.57, despite the fact that the station labels are obviously equivalent.

A token-based similarity measure for strings is the Jaccard Index of their (unique) word sets. Let A be the set of words in station label n_a, and B be the set of words in station label n_b. Then a Jaccard Index based classifier is defined as

$$\text{sim}_\text{JAC}(z_a, z_b) = \frac{|A \cap B|}{|A \cup B|}. \tag{2.40}$$

With the example from above, $\text{sim}_\text{JAC}(z_a, z_b) = 1$.

Best Token Subset-Permutation Similarity

With a token-based Jaccard Index, the impact of each missing token is the same, regardless of its length. It is also not very robust against minor spelling differences or missing location specifiers. We propose a similarity score that tries to combine the advantages of the Jaccard Index (token ordering does not matter) and the edit distance similarity. Given two labels n_a and n_b as well as their unique token sets A and B, this best token

subset-permutation similarity (BTS) can be defined as

$$\mathrm{sim}_{\mathrm{BTS}}(z_a, z_b) = \max\left(\max_{a \in P(A)} sim^*_{\mathrm{ED}}(a, n_b), \max_{b \in P(B)} sim^*_{\mathrm{ED}}(b, n_a)\right), \tag{2.41}$$

where sim^*_{ED} is the edit distance similarity directly on strings and $P(S)$ is the set of all possible unique permutations of S with size $1 \leq n \leq |S|$, concatenated with a space (in other words, all space-separated concatenated possible permutations for each $U \in \mathcal{P}(S)$ with a size > 1, where $\mathcal{P}(S)$ is the power set of S). For example,

$$\begin{aligned} P(\{\text{“Freiburg”}, \text{“Hauptbahnhof”}\}) = \{&\text{“Freiburg”}, \text{“Hauptbahnhof”}, \\ &\text{“Freiburg Hauptbahnhof”}, \\ &\text{“Hauptbahnhof Freiburg”}\}. \end{aligned}$$

The BTS is then the best edit distance similarity between any such permutation of A and n_1, or of any such permutation of B and n_a. An obvious drawback of this measure is its high calculation cost for labels with many tokens, as $|P(S)|$ grows super-exponentially.

TFIDF

A standard method in Information Retrieval to compute relevance scores for terms in documents are TFIDF scores [100], a combination of the term frequency (the number of times a term t occurs in a document d) TF and the document frequency (the number of documents $d \in D$ a term t occurs in) DF. In our case, a term t is a single word (token) in a station label, a document d is a single station identifier label, and the training data (a list of station labels) constitutes the document collection $D \ni d$.

Using the document frequency, we can derive a measure of how rare a term is by taking the quotient of the number of documents $|D|$ and the document frequency. This quotient will be high if the term occurs only in very few documents, and 1 if it occurs in every document. To avoid extremely high values of this inverse document frequency (IDF), it is logarithmically scaled:

$$\mathrm{idf}(t, D) = \log \frac{|D|}{|\{d \in D : t \in d\}|}. \tag{2.42}$$

The TFIDF score of a term, a document and a document collection is then the product of the TF and the IDF. Given a collection of station labels, we can thus derive relevance vectors $\mathbf{v}$ for each label. A vector $\mathbf{v}$ contains TFIDF scores for each token in the label. As a similarity measure $\mathrm{sim}_{\mathrm{TFIDF}}(z_a, z_b)$ we then take the cosine similarity between the

relevance vectors $\mathbf{a}$ and $\mathbf{b}$ for n_a and n_b:

$$\mathrm{sim}_{\mathrm{TFIDF}}(z_a, z_b) = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{a}\|\|\mathbf{b}\|}. \tag{2.43}$$

We note that it may happen that a station label contains tokens that were not encountered in the training data. Then the DF is 0, and the IDF is not well-defined. For such cases we simply set the DF to 1, which means an IDF of $\log |D|$.

Combined Classifiers

To be able to combine the similarity measures described above, we use the following simple approach to map them onto a "non-similar" interval of $[0, 0.5[$ and a "similar" interval of $[0.5, 1]$:

$$sim'(z_a, z_b) = \begin{cases} \frac{1}{2} + \frac{sim(z_a,z_b)-t}{2(1-t)} & \text{if } sim(z_a, z_b) > t, \\ \frac{sim(z_a,z_b)}{2t} & \text{else.} \end{cases} \tag{2.44}$$

Different similarity measures may then for example be combined using soft voting (average) or hard voting (majority vote).

Machine Learning

Determining the optimal threshold values for the similarity measures described above can already be considered a simple form of machine learning with a linear classifier. Using TFIDF scores, we additionally "learn" the significance of tokens, but only on a global level: a token that may be highly significant regionally may appear very often globally, or vice versa. Our evaluation in Section 2.10.1 will give false negative and false positive examples for a TFIDF based classifier where this causes problems. It would also be preferable if our classifier could learn common abbreviations or common spelling mistakes. Also, the presence of certain tokens (for example "Main Station") indicates a major station for which the distance between similar station identifiers may be significantly larger than usual. In this section we develop a classifier which is able to consider these aspects. We extract the following features from our training data of similar and non-similar pairs $\{s_a, s_b\}$ of station identifiers:

1. The great-circle distance between the station identifiers in meters.
2. Let $n_t(z)$ be the number of occurrences of trigram t in the label of station identifier z (for example, $n_{\mathrm{rei}}(z) = 2$ if the label of z is "Freiburg im Breisgau"). For the top-k trigrams in the training dataset, we then add individual features and take $n_t(z_a) - n_t(z_b)$ as a feature value. We pad the tokens with spaces on both sides to encode the beginning and ending of tokens and to ensure that single character tokens are represented by a single token.

Table 2.2: Example feature vectors for three station pairs: (1) "Freiburg im Breisgau Hauptbahnhof" @ (47.9966, 7.8404) vs. "Hauptbahnhof" @ (47.9965, 7.8407). (2) "Okenstraße" @ (48.0105, 7.8545) vs. "Nordstraße" @ (48.0111, 7.8541). (3) "ZOB" @ (47.9959, 7.8405) vs. "Zentraler Omnibusbahnhof, Freiburg im Breisgau" @ (47.9960, 7.8407).

d_m	d_{3g}	x_0	y_0	x_1	y_1	rei	tra	raß	aße	urg	bur	ibu	␣Fr	Fre	eib	rg␣	eis	Bre	sga	isg	simi
24	20	133	196	133	195	-2	0	0	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	**yes**
72	10	133	196	133	195	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**no**
12	47	133	196	133	195	2	1	0	0	1	1	2	1	1	1	1	1	1	1	1	**yes**

3. Let $T(z)$ be the set of *all* trigrams of the again space-padded tokens of station identifier z. To capture difference in trigrams which are not in the top-k, we use the number of trigrams which are present in one station identifiers, but missing in the other as a features. This is $|T(z_a) \cup T(z_b)| - |T(z_a) \cup T(z_a)|$.
4. We take the centroid c of z_a and z_b as a representation for the general region the stations are in. We then take n grids $G_0, \ldots, G_n$, all with the same grid cell width w and height h. The origin for a grid G_i is offsetted by $i(w/n)$ in the x direction, and by $i(h/n)$ in the y direction. The grid cell coordinates of c on each grid G_i are then added as a feature pair (x_i, y_i).

The last feature requires some additional clarification. Consider the example given in Figure 2.19. The intuition behind these interwoven grids is to give the classifier the ability to extract different levels of locality through the interference of selected grids. For example, a combination of grid coordinates (for example $g_0 = (133, 197), g_1 = (133, 196), g_2 = (132, 196)$) may encode a region much smaller than an entire grid cell. Another goal was to soften the effect of the hard grid boundaries.

Table 2.2 gives three example feature vectors. For our evaluation, we trained a random forest classifier on these features.

2.9 Map-Matching Speedup Techniques

We recall from Section 2.5.4 that we determine the optimal hidden state sequence via a shortest path search in a layered graph weighted with log probabilities, which can be done in $O(|V|+|E|)$ as this graph is a DAG. The bottleneck of our map-matching approach is then the calculation of the $|H_i| \cdot |H_{i+1}|$ shortest paths between hidden state layers, which are required to calculate the transition probabilities. As schedule data typically only has to be map-matched in bulk and is updated only a few times a year, the running time of our approach may seem insignificant at first. However, real-word schedule data may consist of millions of vehicle trips. Efficiency thus has to be considered.

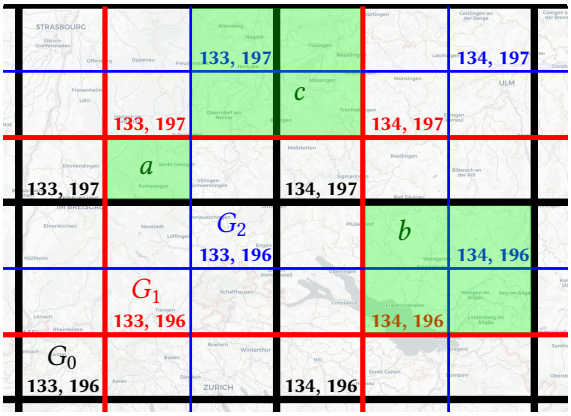


Figure 2.19: Three interwoven grids G_0, G_1 and G_2 used to encode the approximate region of coordinates on three levels of locality. Rectangle a, specified by grid coordinates $g_0 = (133, 197), g_1 = (133, 196), g_2 = (132, 196)$), describes a region on the most local level 0, rectangle b ($g_0 = (134, 196), g_1 = (134, 196)$) a region on level 1, and individual grid cells describe a region on level 2. There is no combination of grid coordinates to describe c.

2.9.1 Faster Layer-to-Layer Shortest Path Calculations

Calculating all shortest paths in the underlying transportation network between sets H_i and H_{i+1} is an instance of a many-to-many shortest path search as described in Section 1.3.3. As mentioned there, a straightforward way to speed up this calculation is to not calculate $|H_i| \cdot |H_{i+1}|$ node-to-node shortest paths using Dijkstra's algorithm, but only $|H_i|$ node-to-many shortest paths, namely for each $h^i \in H_i$ all shortest paths from h^i to H_{i+1}. This can be achieved in $|H_i|$ Dijkstra runs.

Speeding up the calculation of many-to-many shortest paths is an active research topic. However, we show in this section that (somewhat surprisingly), a *single* Dijkstra run is enough to calculate all the *relevant* shortest path costs between H_i and H_{i+1} in our HMM based on shortest travel times. That is, we prove that (1) for some edges from H_i to H_{i+1} in the layered graph we do not require the exact edge weights to find the most likely hidden state sequence, and (2) a single (slightly modified) set-to-many shortest path search between H_i and H_{i+1} using Dijkstra's algorithm is enough to calculate the required weights. As mentioned above, this extends a similar technique from [52].

For simplicity, we first assume that the shortest path costs in the underlying transportation network are equivalent to the transition edge costs in the layered graph.

Lemma 2.6. *Let H_i be the set of all nodes corresponding to hidden states of layer i. By c_j^i we denote the cost of the shortest path from the start node q_0 to node $h_j^i \in H_i$. Assume that all c^i for H_i are known. Then for each $h_k^{i+1} \in H_{i+1}$, $c_k^{i+1} = \min\{c_j^i + w(h_j^i, h_k^{i+1}) \mid h_j^i \in H^i\}$, where $w(h_j^i, h_k^{i+1})$ is the weight of edge (h_j^i, h_k^{i+1}). In other words, the cost c_k^{i+1} of the shortest path from q_0 to h_k^{i+1} will be the smallest sum of an edge weight back to layer i and the known shortest path cost from q_0 to corresponding node in layer i.*

Proof. The proof is straightforward: per the construction of c_k^{i+1}, there cannot be any cheaper path from q_0. □

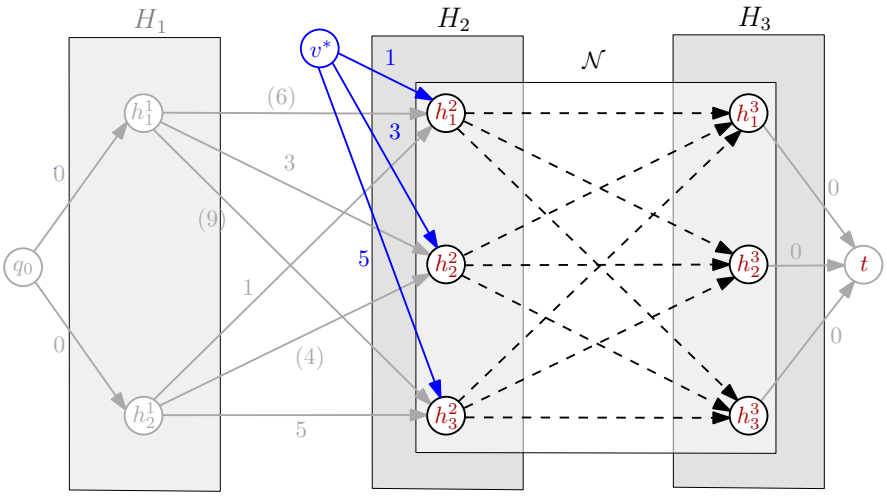


Figure 2.20: Calculation of the required transition shortest path costs between two layers H_2 and H_3 through the transportation network $\mathcal{N}$, under the assumption that we ignore emission probabilities. We add a virtual node $v*$ connected via virtual edges to all $h^2 \in H_2$ and initialize their edge weights to the shortest path cost from q_0 to respective h^2. The edge weights of the previous transitions from layer H_1 to layer H_2 that were not required to calculate the shortest path to the respective $h^2 \in H_2$ are given in brackets, and the required edge weights for the transitions from H_2 to H_3 can be found by a single Dijkstra run on the transportation network $\mathcal{N}$ from v^* to H_3.

This means that for some node $h_k^{i+1} \in H_{i+1}$, we only require the cost of the shortest transitional path starting at *some* node c_j^i in H_i (offsetted by its known shortest path cost c_j^i) in the underlying transportation network to determine the shortest path cost c_k^{i+1} in our HMM layer graph.

Lemma 2.7. *For each node $h_k^{i+1} \in H_{i+1}$, we can calculate the shortest path p_k from H_i to h_k^{i+1} for which $c(p_k) + c_j^i$ is minimal with a single run of Dijkstra's algorithm if we initialize the priority queue with each $h_j^i \in H_i$ and its known shortest path cost c_j^i as a cost label.*

Proof. The proof is similar to the correctness proof of Lemma 1.3. We add a new virtual node v^*, connect it to each $h_j^i \in H_i$ with a virtual edge e_j and set the corresponding edge weight to c_j^i. A single run of Dijkstra's algorithm from v^* then yields for each node $h_k^{i+1} \in H_{i+1}$ a path p_k which consists of a virtual edge e_j with weight c_j^i and the shortest path from h_k^{i+1} (if it would not be the shortest path, then p would not be a shortest path). This path p_k thus minimizes $c(p_k) + c_j^i$. Adding each $h_j^i \in H_i$ to the priority queue with cost label c_j^i implicitly constructs both v^* and the virtual edges. □

Figure 2.20 gives an example.

So far, our assumption was that the edge weights in the layered graph are equivalent to the shortest path cost of the corresponding transition. This is not true in general. First, we have to consider the costs associated with the emission probabilities for the $h^{i+1} \in H_{i+1}$. Luckily, we can simply add the emission weight associated with a hidden state h^i to its offset cost c_j^i. Note that this means that we have to manually add the emission weights for the hidden states of the last layer later on.

Second, we note that the transition probability does not directly depend on the shortest path costs in the underlying transportation network, but is a function of it. Let now c_j^i again be the shortest path cost through the layered graph to hidden state node h_j^i, let e_j^i be the emission probability of h_j^i (in log space), let d be the shortest path cost from h_j^i to h_k^{i+1}, and let $w(d)$ be the transition probability from h_j^i to h_k^{i+1} based on the shortest path cost (also in log space). The Dijkstra run above then yields for each hidden state h_k^{i+1} the cost $c_k^{i+1} = \min\{c_j^i + e_j^i + d \mid h_j^i \in H^i\}$ (note that we can easily extract the value of d from the calculated shortest path, as $c_j^i + e_j^i$ is completely contained in the virtual edge). To be sure that this indeed corresponds to the optimal path if we would not consider d, but $w(d)$, w must be monotonic, that is for two shortest path lengths d_1 and d_2, it must hold that

$$d_1 < d_2 \Leftrightarrow w(d_1) < w(d_2). \tag{2.45}$$

For the transition probabilities described in Section 2.5.2 which were modeled as an exponential distribution, this is obviously the case. Using this model, we can therefore calculate the most likely hidden state sequence for m observations in $O(m(|E_\mathcal{N}| + |V_\mathcal{N}| \log |V_\mathcal{N}|))$, where $E_\mathcal{N}$ and $V_\mathcal{N}$ are the edges and nodes of the underlying transportation network $\mathcal{N}$. The naive method of calculating $H_i \times H_{i+1}$ node-to-node shortest paths would have required an asymptotic running time of $O(mn^2(|E_\mathcal{N}| + |V_\mathcal{N}| \log |V_\mathcal{N}|))$. The straightforward optimization of only calculating n node-to-many shortest path has an asymptotic running time of $O(mn(|E_\mathcal{N}| + |V_\mathcal{N}| \log |V_\mathcal{N}|))$. We will evaluate the effects of this technique in Section 2.10.3.

2.9.2 A^* Heuristic

A^* heuristics as described in Section 1.3.3 are typically given for single target nodes, but it is easy to see that they can also be used in multi-target scenarios (one-to-set, set-to-set, one-to-many, or set-to-many). Their efficiency, however, greatly depends on the characteristics of the target set.

Lemma 2.8. *Let T be a set of nodes. We add a virtual node ω and edges (t, ω) with $w(t, \omega) = 0$ for each $t \in T$. Let h then be a heuristic function for target node ω. If h is an admissible heuristic for ω, it is an admissible heuristic function for each $t \in T$. If h is a consistent heuristic for ω, it is a consistent heuristic for each $t \in T$.*

Proof. For h to be admissible for each $t \in T$, the following must hold: $\forall u \in V, \forall t \in T : h(u) \leq c(u,t)$, where $c(u,t)$ is the shortest path cost from u to t. As h is admissible for ω, it holds that $\forall u \in V : h(u) \leq c(u,\omega)$. But as $w(t,\omega) = 0 \forall t \in T$, $c(u,\omega) = c(u,t) \forall t \in T$, it also holds for each $t \in T$. For h to be consistent for each $t \in T$, the following must hold for each pair of nodes v,u: $\forall t \in T : h(v) \leq w((v,u)) + h(u)$. But this is already the case as h was assumed to be consistent. □

Assume now that we are searching for the shortest path(s) from some node s into a set of target nodes T in a network graph $\mathcal{N}$ for some vehicle type. Let C be the connected component of $\mathcal{N}$ in which s lies and note that any target node t must also be contained in C, or otherwise the shortest path does not exist.

Let now $v_{\max}$ be the maximum possible speed in C, let z be a virtual node positioned at the centroid of all t, and let $R = \max\{d(t,z) \mid t \in T\}$ be the maximum meter distance between z and any t ($d(u,v)$ gives the meter distance between two nodes u and v). We then use the following simple heuristic function:

$$h(v) = \max\left\{0, \frac{d(v,z) - R}{v_{\max}}\right\}. \tag{2.46}$$

Lemma 2.9. *$h(v)$ is admissible for all $t \in T$, that is $\forall v \in V : h(v) \leq c(v,s)$*

Proof. Recall that we model our edge weight function on the travel time and that the punishment for no matching line as described in Section 2.7.1 *increases* travel times. Let $c(u,v)$ be the shortest path cost from u to v. Then

$$\begin{aligned} c(u,t) &\geq \frac{d(v,t)}{v_{\max}} && (2.47)\\ &\geq \frac{d(v,z) - R}{v_{\max}} && (2.48)\\ &\geq \max\left\{0, \frac{d(v,z) - R}{v_{\max}}\right\} = h(v). && (2.49) \end{aligned}$$

□

Lemma 2.10. *$h(v)$ is consistent, that is $h(v) \leq w((v,u)) + h(u)$ for adjacent nodes v,u.*

Proof. We observe that $\frac{d(v,u)}{v_{\max}} \leq w((v,u))$ and prove the stronger condition

$$\begin{aligned} & h(v) \leq \frac{d(v,u)}{v_{\max}} + h(u) && (2.50)\\ \Leftrightarrow \quad & \max\left\{0, \frac{d(v,z) - R}{v_{\max}}\right\} \leq \frac{d(v,u)}{v_{\max}} + \max\left\{0, \frac{d(u,z) - R}{v_{\max}}\right\} && (2.51)\\ \Leftrightarrow \quad & \max\{0, d(v,z) - R\} \leq d(v,u) + \max\{0, d(u,z) - R\}. && (2.52) \end{aligned}$$

There are now 4 cases: if only the left-hand $d(v,z) - R$ falls below zero, we get $0 \leq d(v,u) + d(u,z) - R$, which is obviously always true as we assumed $d(u,z) - R \geq 0$. If only the right-hand $d(u,z) - R$ falls below zero, then u is either inside the circle of radius R around z, or on the border of it. We get $d(v,z) - R \leq d(v,u)$, which is again true as $d(v,z) - R$ is exactly the distance from v to this circle. If both the right-hand $d(u,z) - R$ and the left-hand $d(v,z) - R$ fall below zero, we get $0 \leq d(v,u)$, which is also always true. If neither fall below zero, we have

$$d(v,z) - R \leq d(v,u) + d(u,z) - R \tag{2.53}$$

$$\Leftrightarrow \quad d(v,z) \leq d(v,u) + d(u,z), \tag{2.54}$$

which is true as the meter distance between nodes (we use the haversine formula to calculate them) satisfies the triangle inequality. □

As $h(v)$ is hence consistent, the A^* algorithm will only expand a node u exactly once, and as $h(v) \geq 0$ and $c(u,v) \geq 0$, all nodes relaxed while expanding u will get a new label that is higher than or equal to the last extracted label of u. The priorities extracted from our priority queue Q thus form a monotonically increasing sequence, and we may therefore use a monotone priority queue to get some additional performance gains. In our reference implementation, we used a priority queue based on a radix heap [3].

The efficiency of this heuristic greatly depends on the average distance of visited nodes to the circle with radius R around z, and thus on the maximum distance for candidate nodes around the sample point. If all visited nodes are inside the circle, the heuristic will always be 0, at which point we will be running a raw Dijkstra (with the additional overhead of calculating $h(v)$ for each relaxation). This might for example be the case for very short bus routes, where most of the source hop nodes are already inside this circle. On the other hand, for longer routes, we expect this blurring effect to be insignificant (this will be confirmed by our experimental evaluation in Section 2.10.3).

2.9.3 Early Stopping

A straightforward speedup technique is to not further expand nodes labeled with a cost above some threshold t during the Dijkstra run. For example, if a target node candidate set H_i contains some node h which is in the same connected component as the source node, but is only reachable through a complicated detour, large parts of the graph will be explored to find the shortest path to h, only for it to later yield a transition probability which is orders of magnitudes smaller than other transition probability into H.

As we are dealing with schedule data, we already have a rough upper bound on the time cost of the shortest path - for a stop transition from $i-1$ to i in some a trip T, this is simply the difference $\Delta_i = t_i^{\downarrow} - t_{i-1}^{\uparrow}$ between the arrival time at the stop i and the departure time at the last stop $i-1$. Simply taking this time difference Δ_i as the threshold

t would be too aggressive, as schedule times are usually given with a granularity of one minute, rounded to the nearest minute, and the scheduled travel time between two nearby stops is often 0 for urban bus or rail routes. Another problem is that we might also underestimate edge travel times. In our reference implementation, we used $t = 60 + 3\Delta_i$ seconds as a threshold.

2.9.4 Bulk Processing

In this section, we describe several speedup techniques that use the fact that schedule data is usually map-matched in bulks. We will consider the trivial speedup technique of matching two trips that only differ in their starting time offset *once* as given and therefore assume that all trips are unique in this regard.

Shortest-Path Caching

Even if we consider trips to be unique, many scheduled trips will typically share station-to-station segments. Thus, many of the shortest paths required to determine the transition probabilities will occur over and over again during the map-matching process. A straightforward idea is to only compute their cost once, store it, and use the cached result later on. We use this approach as a baseline speedup technique in our evaluation.

Faster Bulk Processing via Trip Tries

Shortest-path caching is orthogonal to the technique described in Section 2.9.1, where we explicitly avoided calculating too many transitional shortest paths. If we recall our approach there, we realize that to compute the required edge weights between H_i and H_{i+1}, the only used information external to the currently processed layers was the shortest path costs c_j^i through the layered graph to nodes $h_j^i \in H_i$. But if two trips have the same attributes, and of they serve the same stations up until s_i, the c_j^i will be exactly the same, regardless of the further course of the trip. We may thus re-use them. To efficiently calculate, store, and process prefix overlaps of trips, we first compute a forest of trip prefix trees, which we call trip tries, and perform the map-matching on them.

Definition 2.10 (Trip Trie)**.** Let $B = (S, \mathcal{L}, \Lambda, \mathcal{T})$ be a Bag of Trips according to Definition 1.1, and let $\mathcal{T}' \subseteq \mathcal{T}$ be a subset of trips in which for each pair $t_1, t_2 \in \mathcal{T}$ it holds that $l_1 = l_2$ (that is, they all have the same line). A trip trie for $\mathcal{T}'$ is a prefix tree in which each node v is labeled with a station $s \in S$. Each node v is additionally labeled with trips $\mathcal{T}_v' \subseteq \mathcal{T}'$, and for each $t \in \mathcal{T}_v'$ the following holds: the station labels of the (ordered) nodes on the path from v to the tree root form a prefix of the station sequence Σ_t.

Figure 2.21 gives an example of a trip trie for a typical (simplified) bus line with 6 line course variants, served by vehicle trips $t_1, t_2, t_3, t_4, t_5, t_6$. Although trips t_2, t_3, t_4, and

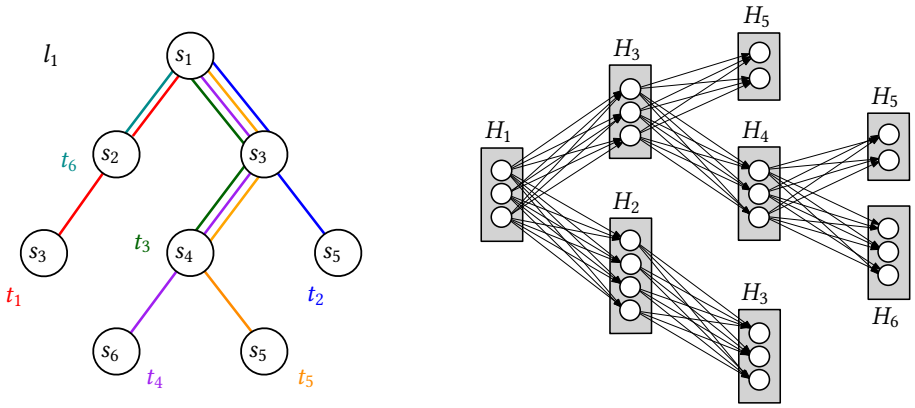


Figure 2.21: Left: A trip trie for a subset $\mathcal{T}' = \{t_1, t_2, t_3, t_6, t_5, t_6\} \subseteq \mathcal{T}$ in which all trips have line l_1. Each node is labeled with a scheduled station and (optionally) a trip ending at this node. Note that the same station may appear in multiple nodes, as is the case for s_3 and s_5 here. Right: Schematic view of the layered graph corresponding to our hidden Markov model, white nodes are the hidden states (the station candidates), and arrows denote hidden state transitions whose weights depend on the shortest path connecting the station candidates.

t_5 have different stops, we can not only re-use the shortest path distances through the transportation network $\mathcal{N}$ for the transitions from H_1 to H_3 and from H_3 to H_4, we may even apply the speedup technique from Section 2.9.1. For real-world schedule data with many different lines, we typically have to build a substantial number of trip tries.

Definition 2.11 (Trip Trie Forest)**.** Given a Bag of Trips B, a trip trie forest contains a trip tree for each line $l \in \mathcal{L}$.

In this definition, the tip trie ignores travel times, but it is easy to see that we can incorporate them by not only labeling a tree node v with a station s, but with a pair (s, δ), where δ is the scheduled travel time from the parent node of v to v.

Node-Candidate Predetermination

It is also evident that we do not need to determine the hidden states when we build the trip trie - given a station s and a transportation network $\mathcal{N}$, the edge candidates and their emission probabilities will always be the same, regardless of the scheduled stops of a trip or the line attributes. We can therefore determine all candidates for all stops once, prior to building the trip trie.

2.10 Experimental Evaluation

We evaluated both our map-matching approach and the advanced station similarity classification on several real-world datasets. In Section 2.10.1, we first present the evaluation results of the station similarity classification methods discussed in Section 2.8. In Section 2.10.2, we then evaluate variants of our map-matching approach against a baseline and the widely-used approach from [109]. In particular, we also measure the impact of the station similarity classification on the map-matching quality. In Section 2.10.3, we then evaluate the overall speed of our approach and the impact of the speedup techniques described in the previous section.

2.10.1 Quality of the Station Similarity Classification Methods

This section gives an overview over the results of an extensive evaluation of station similarity classification methods on two ground truth datasets extracted from OSM, with 2.1 million and 13.6 million station identifier pairs, respectively. Preliminary results have been previously published in [15] and formed the basis of a quality assessment tool for public transportation stations in OSM (*staty*[6]), which has been described in [16]. For this work, we re-ran the evaluation on updated OSM data. The ground truth datasets, pretrained models and a classification playground are available online[7], as is the evaluation setup[8].

Evaluation Setup

We built two ground truth datasets from OSM data for the following regions: the British Isles (BI), and Germany, Austria, and Switzerland (DACH). The dataset dimensions are given in Table 2.3. Our approach was as follows: for each node describing a public transportation stop or station (we filter for all key/value pairs describing station nodes for various methods of transportation described in the OSM wiki[9]), we extracted labels given in the OSM attributes `name`, `ref_name`, `uic_name`, `official_name`, `alt_name`, `loc_name`, `reg_name`, `short_name`, and `gtfs_name`. Each unique label together with the node's latitude/longitude position was then added as a station identifier. Two station identifiers z_a and z_b that were within 1,000 meters were added as *similar* to the ground truth data if either: (1) z_a and z_b came from the same node n. (2) z_a and z_b came from different nodes n_a and n_b, but n_a and n_b were part of the same OSM relation `public_transport=stop_area` (which groups different parts of a larger public transport stop[10]).

[6] https://staty.cs.uni-freiburg.de/
[7] https://staty.cs.uni-freiburg.de/datasets/
[8] https://github.com/ad-freiburg/statsimi-eval
[9] https://wiki.openstreetmap.org/wiki/Public_transport
[10] https://wiki.openstreetmap.org/wiki/Tag:public_transport%3Dstop_area

Table 2.3: Dimensions of our ground-truth datasets for the British Isles (BI) and Germany, Austria, and Switzerland (DACH). N is the number of stations in the OSM data, G the number of station group relations, N' is the number of stations without any group relation, $|s|$ the number of *unique* station identifiers, g is the average group relation size, d^+ is the average distance (in meters) between positive ground truth pairs, K^- is the number of extracted pairs that were not similar, and K^+ is the number of similar pairs. The numbers in this table are without the spicing described in Section 2.10.1.

	N	G	N'	$\|s\|$	g	d^+	K^-	K^+	$\boldsymbol{K}$
BI	270k	15k	234k	261k	3.7	56.7	1.7M	0.4M	**2.1M**
DACH	679k	102k	350k	875k	5	46.1	11.1M	2.6M	**13.6M**

Two station identifiers z_a and z_b within 1,000 meters were added as *not similar* to the ground truth data if z_a and z_b came from different nodes n_a and n_b, but n_a and n_b were members of different OSM relations `public_transport=stop_area`. If in such a situation one or both of n_a and n_b were not part of any `public_transport=stop_area` relation, we did not add $\{z_a, z_b\}$ as a non-similar pair. The reason is that forgetting to group similar station nodes by a `public_transport=stop_area` relation is common in OSM. We ignored station identifier pairs that were more than 1,000 meters away, as we could not find any similar station identifier pair with a distance greater than 1,000 meters in our OSM datasets, except for some obvious mapping mistakes.

For all such ground truth pairs, we additionally performed the following sanity checks: if a pair $\{z_a, z_b\}$ was considered not similar, but z_a and z_b had *exactly* the same label and were within 250 meters, we ignored this pair. If z_a and z_b were in different `public_transport=stop_area` relations r_a and r_b, but r_a and r_b were themselves in a relation `public_transport=stop_area_group` (seldomly used to add an additional hierarchical level for very large stations), we also ignored this pair.

As OSM data contains only very few obvious mistakes, we spiced our ground truth dataset by randomly adding obvious "not similar" pairs. With probability $p = 0.5$, we selected for a station identifier z_a from the OSM data 5 random station identifiers $\{z_b^1, z_b^2, z_b^3, z_b^4, z_b^5\}$ which were more than 1,000 meters away from s (so that we could be sure that they were not similar). For each such z_b, we added a new station identifier z'_b which had the same label as z_b, but was given a new random coordinate within 100 meters of s_a. The pair $\{z_a, z'_b\}$ was then added as "not similar". To also simulate coordinate imprecision, we randomly selected similar station pairs and added some Gaussian noise to the coordinates of one station (we used a standard deviation of 100 meters).

The ground truth dataset was then divided into a random selection of 20% of the data as a training set, and a random selection of 80% of the data as a testing set. The immense size of the ground truth datasets allowed for such a small training set, and preliminary experiments showed only minimal gains when larger training sets were

Table 2.4: Station similarity classification evaluation results.

	BI				DACH			
method	t	prec.	rec.	F1	t	prec.	rec.	F1
P	100 m	0.66	0.93	0.77	125 m	0.4	0.97	0.57
ED	0.85	0.99	0.85	**0.91**	0.85	0.99	0.66	0.79
PED	0.85	0.93	0.88	0.91	0.9	0.92	0.73	0.81
J	0.9	0.98	0.85	0.91	0.85	0.93	0.7	0.8
JW	0.9	0.93	0.88	0.91	0.9	0.9	0.71	0.8
JAC	0.65	0.96	0.86	0.91	0.45	0.84	0.88	0.86
BTS	0.85	0.91	0.9	0.91	0.85	0.92	0.92	**0.92**
TFIDF	0.99	0.99	0.83	0.9	0.7	0.9	0.84	0.87
P+ED	40 m + 0.6	0.96	0.9	0.93	40 m + 0.55	0.9	0.82	0.86
P+JAC	80 m + 0.55	0.95	0.93	**0.94**	90 m + 0.45	0.92	0.93	0.92
P+BTS	10 m + 0.5	0.93	0.9	0.91	20 m + 0.65	0.92	0.91	0.92
P+TFIDF	40 m + 0.2	0.95	0.93	0.94	60 m + 0.5	0.94	0.93	**0.94**
RF	—	> 0.99	0.98	$>$ **0.99**	—	> 0.99	> 0.99	$>$ **0.99**

chosen. All classifiers were then trained on the training datasets, and evaluated against the test datasets (for the similarity measure based classifiers, training meant finding the optimal threshold values).

Our evaluation metrics were based on the number of true positives (TP), the number of true negatives (TN), the number of false positives (FP), and the number of false negatives (FN). Namely, we used precision ($\frac{\text{TP}}{\text{TP+FP}}$), recall ($\frac{\text{TP}}{\text{TP+FN}}$), and F1 scores, where the latter is the harmonic mean between precision and recall.

Results

Our results, averaged from 5 runs, are reported in Table 2.4. We evaluated each standalone classifier on both test datasets. Using the technique described in Section 2.8.2, we also evaluated the following combinations of classifiers: P+ED, P+JAC, P+BTS, and P+TFIDF, and a random forest (RF) based classifier using the features described in Section 2.8.2 (we used the random forest classifier provided by the Python `scikit-learn` library with default parameters). For the feature vectors, we used the top 2,500 trigrams and 2 grids G_0 and G_1, with cell widths and heights chosen such that the earth was covered by 256×256 cells.

For both datasets, the random forest based classifier (RF) clearly outperformed each other classifier, achieving F1 scores > 0.99. For the combined classifiers, two winners emerged: P+TFIDF and P+JAC, but only by a small margin to P+ED and P+BTS. In particular, a simple combination of the geographical distance and the meter distance (P+ED)

FN	"Parkweg" @ (52.0149, 7.2051) "Rosendahl, Osterwick, Parkweg" @ (52.0149, 7.2051)
FP	"Bruck an der Mur" @ (47.4136, 15.2793) "Bruck an der Mur, Waldweg" @ (47.4185, 15.2736)

Figure 2.22: A typical false negative and a typical false positive given by a Jaccard index based classifier. The location specifiers "Rosendahl, Osterwick" prevents a positive classification in the first example. In the second example, a train station labeled after a town was positively matched against a bus stop "Waldweg" provided with the town name as a location specifier.

FN	"Bromley-By-Bow Platform 2" @ (51.5248, −0.0115) "Bromley By Bow Station" @ (51.5234, −0.0121)
FP	"Clapton Girls' Academy" @ (1.5539, −0.0537) "Clapton" @ (51.5617, −0.0568)

Figure 2.23: A typical false negative and a typical false positive given by a PED based classifier.

already achieved an F1 score of 0.93 on the BI dataset, but only of 0.86 on the DACH dataset. Upon manual investigation, we found that station identifier names were much more uniform and standardized on the British Isles than they were in German-speaking countries. This was also reflected in the high performance of the ED standalone classifier on the BI dataset, which already achieved an F1 score of 0.91, but only 0.79 on the DACH dataset. Because of this station name uniformity, all similarity measures based on string similarity performed equally well on the BI dataset, achieving F1 scores between 0.9 and 0.91. For these measures, the differences on the DACH dataset were much greater: F1 scores ranged from 0.79 to 0.92, with the BTS classifier achieving the best result.

The reasons for false positives and false negatives for the standard string similarity based classifier were as expected. Figure 2.23 gives an example of a typical FN and FP for the PED classifier and Figure 2.22 shows typical such examples for the JAC classifier. For the TFIDF based classifier, the reasons for erroneous classification were more interesting and subtle. Consider for example Figure 2.24. Here, one problem was that the TFIDF classifier learned that some tokens have very little relevance on a global (or, in this case, national) level, but these tokens had high relevance on a regional or local level. These tokens were typically locations which are present in most villages, like cemeteries, schools, churches, or town halls. On the other hand, the classifier had a tendency to give village and town names which were unique nationally too much significance.

Finally, we found the following main causes for the remaining false negatives and false positives of the random forest classifier: (1) extreme outliers, (2) stations at street

FN	"Auerbach (Karlsbad), Rosenweg" @ (48.9161, 8.5341) "Rosenweg" @ (48.9160, 8.5343)
FP	"Cottbus, Kiekebusch Alte Schule" @ (51.7215, 14.3646) "Kiekebusch Friedhof, Cottbus" @ (51.7179, 14.3672)

Figure 2.24: A typical false negative and a typical false positive given by a TFIDF based classifier. Because the input DACH dataset was large, the TFIDF scores gave the nationally common token "Schule" (school) and "Friedhof" (cemetery) only little significance, although they are highly significant locally. On the other hand, the TFIDF scores gave the village name "Auerbach" which is rare nationally, too much significance, although many stations in this region are prefixed by it.

FN	"Little Ilford School" @ (51.5483, 0.0577) "Church Road" @ (51.5479, 0.0569)
FP	"Galsworthy Road/Moonshine Lane" @ (53.4178, −1.4808) "Moonshine Lane - Galsworthy Road" @ (53.4178, −1.4803)

Figure 2.25: A typical false negative and a typical false positive for our RF based classifier. For the false negative, our model actually found a mapping mistake in OSM. The false positive are two stations at the same intersection, but in different streets. As the RF classifier does not consider the ordering of trigrams, they were classified as similar.

intersections as shown in Figure 2.25, (3) ambiguous cases, and (4) errors in the ground truth data. As mentioned above, the last point motivated the development of the OSM quality assessment tool *staty*[11], which provides suggestions for grouping and renaming stations based on the classification results of our RF classifier [16].

Choosing the Best Classifier for Map-Matching For our map-matching task, there are three criteria a classifier should meet: (1) it must be fast, (2) it must be accurate, and (3) it should require minimal precomputation, as we want to map-match schedule data without first training a classifier or learning TFIDF scores from some external dataset. Based on these criteria, the P+JAC classifier was the clear winner. Threshold values were similar on both datasets, it does not require any precomputation, and it can be efficiently computed (the other contender requiring no precomputation, P+BTS, turned out to be too costly to compute during the map-matching process). We consequently used a P+JAC classifier with the averaged thresholds from both test datasets, that is 85 meters and a Jaccard index of 0.5. The impact of this station similarity classification on the map-matching process will be evaluated in the next section.

[11] `https://staty.cs.uni-freiburg.de/`

2.10.2 Quality of Our Map-Matching Approach

In this section, we evaluate the quality of several variants of our map-matching approach: a raw version without considering line or station metadata (OURS), a version with an additional penalty for station candidates classified as "not similar" (OURS+SM), a version with an additional time penalty for unmatched transit lines in the input data (OURS+LM), and a version with both the unmatched station candidate and the unmatched transit line penalty (OURS+SM+LM). We compare them against the best baseline from Section 2.4, namely the global set-to-set routing approach based on the approach from [52] (G-STS), and a state-of-the-art map-matching technique from [109] (DIST-DIFF, also based on a HMM). DIST-DIFF is for example used in the GraphHopper[12] and the OSRM routing engines[13]. DIST-DIFF does not use stations as hidden states, but road segments, and uses a normal distribution for the emission probabilities. As mentioned above, the transition probabilities are based on the difference between the sample point distance, and the meter length of the shortest path.

We used the following public transit schedules for our evaluation: the bus and tram network of Vitoria-Gasteiz, the bus, tram, rail, ferry, and funicular network of Zurich, the bus, tram, rail, and ferry network of Seattle, the bus, tram, and subway network of Vienna, the bus, tram, rail, and subway network of Paris, the entire public transit network of Switzerland, and the entire public transit network of Germany. The datasets were given in the General Transit Feed Specification (GTFS) format [76]. Table 2.5 gives an overview over their dimensions.

Evaluation Setup

For the quality evaluation of our map-matching approaches, we removed existing ground truth shape information from the testing dataset and map-matched this stripped version. The results were then compared to the original ground truth paths using the metrics described below.

Finding evaluation datasets which gave a good ground truth to measure the quality proved to be difficult for several reasons. First, GTFS data is often published with incomplete shape information, or none at all (as our survey in Section 2.1.2 clearly showed). Second, many GTFS datasets lack the optional `shape_dist_traveled` field, which gives each vehicle stop a distinct progression on the shape. Without this field, it is not possible to reliably find out the stop position on the shape (because the stop's coordinates are typically placed at the station centroid, snapping the stop to the shape is prone to errors). Even if shape data with the `shape_dist_traveled` field is available, the quality is often unsatisfactory, either because the sampling rate is very low, or because shapes are simply incorrect, which would render a quality evaluation meaningless.

[12] `https://github.com/graphhopper/graphhopper`
[13] `https://github.com/Project-OSRM`

Table 2.5: Dimensions of our testing datasets. Under *stations* we give the total number of stations contained in the GTFS feed. Under *trips* we give the total number of contained trips. Under *unique trips* we give the number of trips that only differ in their attributes (MOT, line name) and their station course, but not in the time offset at the first station or the service date. Under *tries* we give the number of trip tries (unique trips sharing common station course prefixes). Under *shapes* we denote whether ground truth shape data was available. The total number of edges over all transportation network graphs for all MOTs is given under $|E|$.

	stations	trips	unique trips	tries	shapes	$\|E\|$
Vit.-Gast. (VG)	338	4.0k	57	37	•	13.4k
Seattle (SE)	6.7k	27.7k	545	408	•	281.6k
Zurich (ZH)	5.6k	96.1k	1.9k	1.1k	•	137.7k
Vienna (V)	4.5k	176.7k	1.1k	664	•	83.9k
Paris (P)	53.7k	307.1k	10.4k	6.3k		830.5k
Switzerland (CH)	39.0k	545.6k	69.8k	42.2k		4.8M
Germany (DE)	501.4k	1.7M	243.5k	109.8k		64.7M

We could not find a large national schedule dataset matching these criteria. We therefore evaluated the quality of our approach on a subset of our evaluation datasets for which good ground-truth data was available: Vitoria-Gasteiz, Seattle, Zurich, and Vienna, the union of which covers bus, streetcar, subway, rail, funicular, and ferry services. For the Switzerland and Germany dataset (both containing the entire nation-wide schedule data for all methods of transportation), and the Paris dataset, we only evaluated the speed of our approach (the quality can for example be inspected manually with our tool TRAVIC[14]). During our experiments, we also noticed that schedule data fitting the criteria described above typically had station coordinates of very high precision. From our experience, this is an exception in real-world schedule datasets (for example, the dataset for Germany only has station centroids as coordinates). To measure the quality of our approaches under imprecise station coordinates as they are typically found in real-world schedule data, we hence also evaluated all of them under Gaussian noise added to the station coordinates.

For the Seattle dataset, we found the coverage of public transit station in OSM to be highly incomplete. We therefore first snapped the stations from the GTFS dataset onto the transportation network. The corresponding station node candidates were slightly penalized during the map-matching process.

To make comparisons fair, we used the same underlying routing engine (implemented by us) for all tested methods. For all methods, we only considered station node

[14] `https://travic.app`

candidates (G-STS, OURS+*) or road segment candidates (DIST-DIFF) that were within a distance of 100 meters to the sample points. For the parameter λ_t of the transition probabilities, we used fixed values: For DIST-DIFF, we fixed λ_t using an estimator suggested in [109]. The estimated parameter corresponded to a mean meter distance of 17.04 meters. For OURS, OURS+SM, OURS+LM, and OURS+SM+LM, we used a λ_t that corresponded to a median travel time between stations of 2 minutes. For the emission probabilities for all HMM based methods, we used the parameters that optimized the average results over all test datasets, under no noise. This will be further explained below. The full evaluation setup can be found online[15].

Evaluation Metrics

We used two metrics to compare a shape found by our approach to the corresponding ground truth: the average Fréchet Distance δ_{aF} and the percentage A_N of ground-truth segments that could not be matched (within a small tolerance) to the corresponding segments in the map-matched shape.

Average Fréchet Distance The Fréchet Distance δ_F [63] between two curves P and Q is a well-established method of measuring their similarity. It is often described as the minimum length of a leash that allows a person and their dog to do a complete walk on P and Q. They can both go forward, stop, but may never backtrack.

In its discrete version δ_{dF} it defines a coupling $C = (p_{a_0}, q_{b_0}), (p_{a_1}, q_{b_1}), ..., (p_{a_n}, q_{b_n})$ of points on $p \in P, q \in Q$ that minimizes $\delta_{dF} = \|C\| = \max_i \|p_{a_i} - q_{b_i}\|$, where $\|p_{a_i} - q_{b_i}\|$ is the euclidean distance. a and b are both monotonously increasing with a step size of at most 1. To approximate δ_F, the sampling rate of both P and Q has to be increased. If the sampling rate goes towards infinity, δ_{dF} converges to δ_F [53].

Both δ_F and δ_{dF} give short outliers significant impact. A metric that better matches our intuitive understanding of curve similarity and which is used as our first evaluation metric is the average Fréchet Distance δ_{aF} [32]. δ_{aF} may be defined as the summed Fréchet Distance $\delta_{sF}(P, Q)$ divided by the total length $\|P\|$ of P: $\delta_{aF}(P, Q) = \delta_{sF}(P, Q)/\|P\|$.

The summed Fréchet Distance δ_{sF} can be approximated as follows [32]:

$$\delta_{sF} \approx \min_C \sum_{(p_{a_i}, q_{b_i}) \in C} \|p_{a_i} - p_{a_i-1}\| \cdot \|p_{a_i} - q_{b_i}\|, \tag{2.55}$$

which is the minimized sum of the distances between individual couples $(p_{a_i}, q_{b_i}) \in C$, weighted by the length of the segment of P defined by p_{a_i} and its predecessor p_{a_i-1}.

[15] https://github.com/ad-freiburg/pfaedle-eval

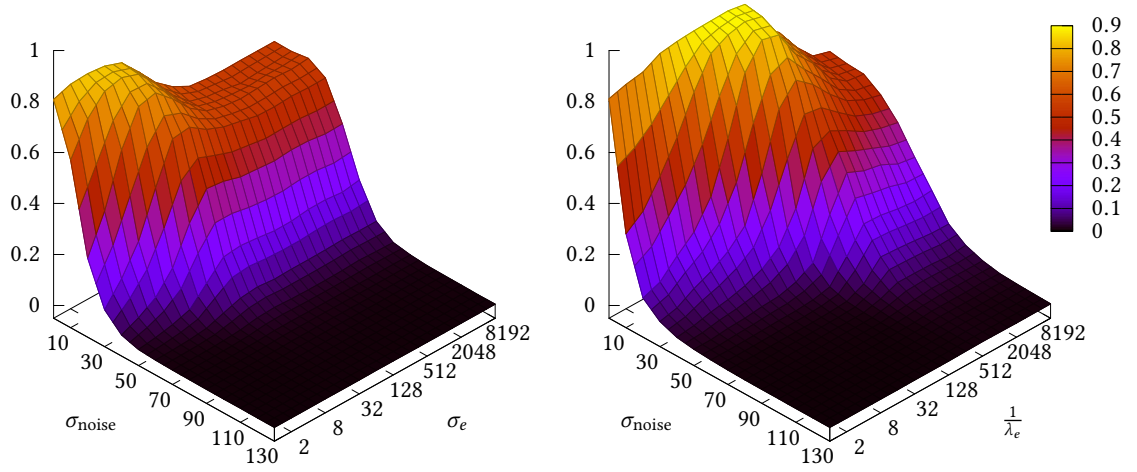


Figure 2.26: Left: A_N^{10} scores for DIST-DIFF under increasing Gaussian noise (with standard deviation σ_{noise}) for different emission probability parameters σ_e. Right: A_N^{10} scores for OURS under increasing Gaussian noise for different transition weights λ_t.

Non-Matched Hops Our second metric is directly based on the discrete Fréchet distance δ_{dF}. For some segment $P_{[a,b]}$ between two stations a and b on the map-matched path P and the corresponding segment $Q_{[a,b]}$ in the ground truth Q, we say that $P_{[a,b]}$ matches $Q_{[a,b]}$ if $\delta_{dF}(P_{[a,b]}, Q_{[a,b]}) \leq 100$ meters (rail platforms and bus stops typically follow the road or tracks for a certain length, with no well-defined stopping position of vehicles, and the threshold value of 100 meters aims to account for that). With the total number of hops N and the number of unmatched hops A, we then define $A_N = A/N$. For example, $A_N = 0.1$ means that 10% of the hops in Q could not be matched to the corresponding hop in P. For our evaluation, we counted the percentage of trips with an A_N score below 0.1. We denote this accuracy measure by A_N^{10}.

Evaluation Results

Figure 2.26 gives the A_N^{10} scores for DIST-DIFF and OURS for various levels of Gaussian noise, under different parameters for the emission probabilities. As described above, we used fixed parameters for the transition probabilities. The scores are averaged over all test datasets. The evaluation was run 10 times, and the final values again averaged.

For further evaluation, we used the emission probability parameters that optimized the average A_N^{10} scores under no noise. These values were a standard deviation σ_e of 4 meters for DIST-DIFF, and a parameter λ_t corresponding to a mean of 128 meters for OURS. For DIST-DIFF, the value of σ_e almost exactly matched the value suggested in [109], which was 4.07 meters.

For OURS+SM, we multiplied the emission probabilities by $w_N = 1.5$ if the scheduled

Table 2.6: A_N^{10} scores (percentage of trips with an A_N score $\leq$ 10%) for all evaluated methods, without noise, on all datasets with ground truth data available. Best results in bold.

	G-STS	DIST-DIFF	OURS	OURS+SM	OURS+LM	OURS+SM+LM
VG	18.4%	59.2%	73.1%	73.1%	73.1%	**73.1%**
SE	0.8%	92.8%	97.3%	97.3%	98.1%	**98.1%**
ZH	28.3%	84.5%	89.7%	89.9%	91.7%	**92.0%**
V	24.1%	97.9%	**98.1%**	97.9%	97.8%	97.6%
avg	17.9%	83.6%	89.5%	89.6%	90.2%	**90.2%**

station and the candidate station were not classified as similar. For OURS+LM, we added a penalty factor of $w_L = 1.2$ to edge weights with no matching line information on them. OURS+SM+LM used exactly the same penalties.

The results for our quality evaluation without Gaussian noise can be found in Tables 2.6, 2.7. Tables 2.8, and 2.9 give the results under the same setup, but with a Gaussian noise with standard deviation of 30 meters. All experiments were done on an Intel Xeon E5-1650 machine with 6 cores, each with 3.5 GHz. Under Gaussian noise the results were averaged from 50 runs. Our main results can be summarized as follows:

Baseline techniques are unsuitable for practice. The G-STS method, arguably the most elaborate of our baseline techniques as it globally optimizes the path over the station candidates, was clearly inferior on all datasets (without noise). Under the δ_{aF} metric, G-STS performed much better (but still worse than all other tested methods), suggesting that G-STS mainly has a problem with correctly placing stations.

For precise input data, DIST-DIFF is already very good. Without noise, DIST-DIFF performed very well, achieving similar results as OURS on the Vienna dataset. Under the δ_{aF} metric on Vienna, it outperformed all our approaches by a small margin.

DIST-DIFF is less robust against noise than our approach. Under noise, the quality of the DIST-DIFF decreases significantly faster than our approach. This effect can already be seen from Figure 2.26.

Penalties for unsimilar stations or lines help under noise. Without noise, both the penalty for unmatched stations and unmatched lines improved the A_N^{10} scores only very little - for one dataset (Vienna), the results even got slightly worse. The δ_{aF} improved for all datasets, however. Under noise, the penalties greatly improved the results, from 6.3 percentage points for the Zurich dataset to 12.4 percentage points for the Seattle dataset. δ_{aF} improved similarly.

Table 2.7: δ_{aF} scores (average Fréchet distance in meters), averaged over all trips. Best results in bold.

	G-STS	DIST-DIFF	OURS	OURS+SM	OURS+LM	OURS+SM+LM
VG	6.2	8.5	4.4	4.4	3.5	**3.5**
SE	20.4	14.7	10.0	9.7	9.0	**8.7**
ZH	16.0	9.2	8.0	7.9	7.7	**7.6**
V	15.1	**7.5**	7.8	7.8	7.7	7.7
avg	14.4	9.9	7.5	7.5	7.0	**6.9**

Even with no noise, the baseline G-STS method was clearly inferior to all other tested methods under the A_N^{10} score, achieving only an average of 17.9% of trips with an A_N value of under 10%. Under the average Fréchet distance, the difference was less pronounced (Table 2.7), and for the Vitoria-Gasteiz dataset, it scored even better than DIST-DIFF. This suggests that while G-STS is bad at placing stations correctly, the overall vehicle paths are mostly correct. However, OURS, OURS+SM, OURS+LM, and OURS+SM+LM, still clearly outperformed G-STS.

To good performance of DIST-DIFF with no noise under both the A_N^{10} score and the average Fréchet distance can be explained by the high quality of the evaluation datasets and the aggressive emission probability distribution parameter $\sigma_e = 4$. This results in stations being almost always placed at exactly the position where the ground truth expects them, with the corresponding road segments being connected by the shortest path. DIST-DIFF performed particularly well on datasets covering cities with a regular transportation network. For example, both Seattle and Vienna have a grid-like road network with many long straight roads. Here, the DIST-DIFF method shines, as it gives the highest probability to transitional paths with a length close to the sample point distance. This was reflected in the ground-truth data: the median difference between the meter length of the path connecting two stations, and their distance, was only around 1 meter (!) for Seattle. For Vienna, it was around 7 meters. In comparison, it was over 12 meters for Sydney, and around 19 meters for both Vitoria-Gasteiz and Zurich. (The DIST-DIFF method was evaluated on a dataset from Seattle in the original publication [109]).

As G-STS adds no penalty to candidate stations with a larger distance from the sample station, its quality under both metrics decreased only little under Gaussian noise (caused by the slightly different station candidate sets). The quality if the DIST-DIFF approach decreased more quickly under noise than OURS. For a low standard deviation of the emission probability distribution, this was to be expected, as the emission probability largely defines the outcome. But even for higher standard deviations, the quality under noise decreased faster than for OURS, as seen in Figure 2.26. Interestingly, the quality decreased most for the Seattle dataset, with an A_N^{10} score of only 9.9% under

Table 2.8: A_N^{10} scores (percentage of trips with an A_N score $\leq$ 10%) using the same setup as in Table 2.6, but under Gaussian noise with a standard deviation of 30 meters to simulate low-quality input data. Best results in bold.

	G-STS	DIST-DIFF	OURS	OURS+SM	OURS+LM	OURS+SM+LM
VG	15.5%	10.6%	49.3%	55.6%	55.7%	**56.6%**
SE	0.9%	9.9%	52.2%	55.0%	62.3%	**64.6%**
ZH	22.8%	34.8%	67.6%	67.0%	73.4%	**73.9%**
V	23.8%	51.0%	75.6%	77.9%	79.9%	**82.6%**
avg	15.8%	26.6%	61.2%	63.9%	67.8%	**69.4%**

Table 2.9: δ_{aF} scores (average Fréchet distance in meters) under Gaussian noise with a standard deviation of 30 meters. Best results in bold.

	G-STS	DIST-DIFF	OURS	OURS+SM	OURS+LM	OURS+SM+LM
VG	9.1	19.3	6.8	6.0	5.5	**4.9**
SE	20.1	30.9	14.1	13.9	11.6	**11.4**
ZH	16.4	18.2	10.7	10.8	9.6	**9.5**
V	15.5	16.2	10.4	10.4	9.2	**9.2**
avg	15.3	21.5	10.5	10.3	9.0	**8.8**

noise, compared to 92.8% with no noise.

In general, the additional penalty for unsimilar station candidates and unmatched lines gave only moderate quality improvements without noise. On average, the A_N^{10} score under no nose improved only by 0.7 percentage points from OURS to OURS+SM+LM. Under noise, however, it significantly improved from 61.2% to 69.4% on average. This of course depended on the quality of the corresponding metadata in OSM. For example, both the line and station data in OSM was incomplete or seemed outdated for Seattle, explaining the drop of the A_N^{10} score for OURS+SM+LM from 98.1% under no noise, to 64.6% under noise. On the other hand, metadata quality was excellent for Vienna.

2.10.3 Speed

To finally measure the impact of our speedup techniques, we ran our approach with unmatched station penalty and unmatched line penalty (OURS+SM+LM) with the parameters from the previous section on all datasets (including those without ground truth data available) using the following speedup techniques: the baseline technique which calculates *all* shortest paths for a single hop from H_i to H_{i+1} using $|H_i|$ 1-to-$|H_{i+1}|$ shortest path

Table 2.10: Running times of our baseline approach B (clustered equivalent trips and n 1-to-m Dijkstra runs between layers), CA (with shortest path cost caching), SDL (single Dijkstra run between two layers), CA+TR (shortest path cost caching on a trip trie), and SDL+TR (single Dijkstra run between layers on a trip trie). The unstarred versions use a standard Dijkstra implementation, the starred versions use the A^* heuristic described in Section 2.9.2. Best times are printed bold.

	B	CA	SDL	CA+TR	SDL+TR	B*	CA*	SDL*	CA+TR*	SDL+TR*
VG	2.3s	1.3s	0.1s	1.3s	90ms	1.9s	1.2s	0.1s	1.2s	**83ms**
SE	1.6m	1.4m	3.3s	1.3m	2.9s	57.5s	49.3s	2.2s	47.7s	**1.9s**
ZH	35.1s	26.3s	1.8s	25.1s	1.4s	31.1s	22.6s	1.7s	21.8s	**1.3s**
V	42.7s	35.2s	1.9s	30.7s	1.5s	32.4s	26.2s	1.7s	22.7s	**1.4s**
P	18.6m	14.0m	43.7s	13.7m	38.3s	10.2m	7.8m	27.4s	7.5m	**23.7s**
CH	1.5h	39.8m	2.8m	34.6m	2.2m	50.3m	21.2m	1.7m	17.3m	**1.3m**
DE	10.2h	7.6h	23.2m	7.5h	21.0m	3.8h	2.5h	9.9m	2.4h	**8.5m**

calculations with a basic implementation of Dijkstra's algorithm (B), the baseline with additional caching of shortest path costs (CA), the method for calculating all *required* shortest path costs using a single Dijkstra run between hops (SDL), the cache based approach with better bulk processing using trip tries (CA+TR), and the single Dijkstra run approach on trip tries (SDL+TR). For each method we evaluated a raw version, and a starred version using the A^* heuristic described in Section 2.9.2. Note that the baseline version was already only run on unique trips. We did *not* measure the time required to parse the OSM input, as this parser is still preliminary and can so far only handle OSM XML files.

The experiments were run on an Intel Xeon E5-1650 machine with 6 cores, each with 3.5 GHz. Results are reported in Table 2.10.

When compared to the baseline, an additional shortest path cost cache gave only moderate improvements. This can mainly be explained by the relatively small overlaps of unique trips in all test datasets. If a shortest path cost cache was combined with the trip tries technique (CA+TR), the additional effect was only small as any additionally calculation saved by a common prefix of two trips would already have been saved by the cache. The additional speedup of CA+TR when compared to the standalone CA technique can partly be explained by the reduced overhead of the cache bookkeeping. We also only use one shortest path cost cache per thread to avoid the overhead of managing parallel writes to the cache, so some lost cross-thread caching opportunities were salvaged by the trip trie preprocessing.

The effect of the SDL technique was dramatic, allowing us to map-match the entire schedule data for Germany in under 25 minutes when used alone, compared to over 10

hours with the baseline method. An additional calculation of the trip tries (SDL+TR) again only gave moderate improvements due to the relatively low overlap of unique trips.

The A^* heuristic had a stronger impact on datasets covering a large geographical area, bringing for example the baseline running time on Germany down from over 10 hours to only 3.8 hours. On smaller datasets the improvement was much smaller, as expected in Section 2.9.2.

Using the SDL+TR* technique, we were able to completely map-match all test datasets in under 9 minutes. For our largest testing dataset Germany, this meant a map-matching time of around 2 milliseconds per unique trip, without any preprocessing besides the building of the trip tries. We note that this time does not only include the map-matching process, but also the building, simplification, and serialization to disk of the GTFS shapes. Given that a naive baseline approach would require $|H_i| \cdot |H_{i+1}|$ shortest path calculations for a single hop (for the bus network of Germany, the average $|H_i|$ was 21.3, and the average number of stops per trip was 20.9), and given the fact that we use dynamic edge costs based on the routing attributes, we consider this a satisfying result.

2.11 Conclusions and Future Work

This chapter adapted a state-of-the-art map-matching approach using a hidden Markov model to map-match schedule data, where sample points are only given at stations. We described how turn restrictions can both be respected inside single transitions between hidden states, and across hidden state transitions, by routing on the edge-to-node dual graph, with vehicle orientations at candidate nodes encoded by adjacent edges. We showed how metadata present in the underlying transportation network, in particular station and public transit line labels, can be incorporated into this approach. In this context, we evaluated several methods for deciding whether two station identifiers, consistent of a geographical position and a label, are similar. The quality of our raw approach, and with the consideration of the network metadata, was compared to a simple baseline method, and an established map-matching approach using a HMM, showing that our approach already performs best without considering the network metadata. Our evaluation also showed that the network metadata makes the approach more robust against low-quality input data.

Using several speedup techniques, we were able to map-match the entire national schedule data for Germany in 8.5 minutes. Smaller datasets for individual cities were typically map-matched in under 2 seconds. It would nevertheless be interesting to evaluate faster algorithms for the shortest path calculation that lies at the core of our transition probability calculation. Using Contraction Hierarchies (CHs) here seems promising, but a direct application requires a constant cost metric. As our transition probabilities

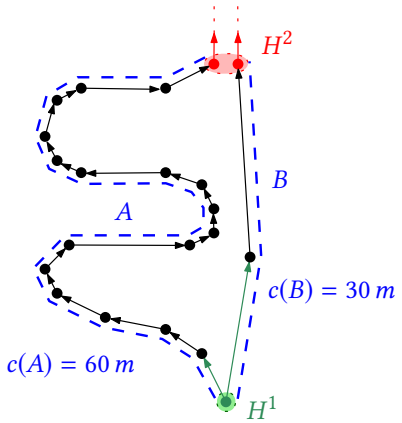


Figure 2.27: Map matching a trip from a station s_1 to s_2, with candidate sets H^1 and H^2. Path A takes much longer (60 minutes) than path B (30 minutes). However, if the scheduled travel time from s_1 to s_2 is around 60 minutes, A is the more likely path. Our approach so far does not use this temporal information.

use the vehicles line names to compute a dynamic edge cost function, a direct application of CHs is not possible. However, techniques exist in the literature for using CHs with a variable cost metric [47].

Our work so far does not use the temporal information present in public transit schedules, but always prefers the route of shortest travel time. However, as Figure 2.27 shows, the fastest route between two station candidates may not necessarily be the most likely one. We already experimented with approaches which based the transition probability on the difference between the schedule travel time, and the real-world travel time, but found the results to be vastly inferior. The obvious reason is that the travel time given in schedule data is extremely unreliable: it is typically rounded to the nearest minute (sometimes resulting in scheduled travel times of 0 between close stations), and often includes buffer times to improve punctuality. Additionally, the road travel times extracted from the transportation network are only estimates, and may differ greatly from the real-world travel times depending on the time of day or even the vehicle type (a regional train, for example, travels a lot slower than a high-speed train).

Finally, our approach may also be a valuable pre-processing step for crowdsourcing public transit vehicle information. Given the most likely path a vehicle will take between stations through a network, it is straightforward to interpolate the approximate position at time t from the schedule data. This would enable us to do map-matching on the temporal domain: given the timestamped location measurements of a smartphone over the last minutes, we would then ask for the most likely public transit vehicle the owner of the smartphone is currently traveling in. Once a sufficiently probable coupling between a smartphone and a public transit vehicle is established this way, the smartphone can be used to provide a precise location of the vehicle. Together with the schedule data, public transit vehicle delay information could thus be crowdsourced from passenger smartphones.

Chapter 3

Line Graph Construction

The previous chapter described how geographical line courses can be extracted from schedule data and a physical transportation network. This map-matching approach yielded a bag of trips $(S, \mathcal{L}, \Lambda, \mathcal{T})$ and for each consecutive stop pair (s_i, s_{i+1}) of a trip $T \in \mathcal{T}$ a polyline describing the geographical course between s_i and s_{i+1}. This already has a natural graph representation fitting the line graph definition from Section 1.3.2: for each $s_i \in S$, add a node v_i to the line graph G. For each consecutive stop pair (s_i, s_{i+1}) of trip T, add an edge $\{v_i, v_{i+1}\}$, labeled with the trip's line L and the hop geometry. This line graph may be a multigraph, as there are typically hundreds of trips between two stops. If *all* edges from v_i to v_{i+1} have exactly the same geometry (within some tolerance), it is straightforward to transform this multigraph into a graph - simply combine all edges into a single edge labeled with (1) the shared geometry (or maybe the average geometry), and (2) the union of all edge lines. There are three problems with this approach:

1. The line graph may be degenerate: if each (unique) trip in the bag of trip has distinct stops, then G will be a collection of path graphs.
2. Two edges e and f connecting nodes u and v may have different geometries.
3. These might still *overlap*.

Consider Figure 3.1, left. Even if stations A, B, and C would be distinct nodes, problems 2 and 3 persist. If we solve problem 2 by merging the edges labeled with A and B, and the edges labeled with D and C (Figure 3.1, middle), there would still be considerable overlap in the final map rendering (Figure 3.1, right). In this section, our aim is thus to construct a so-called *free* line graph - suitable for rendering - from either a bag of single line geometries (the degenerated case), or from an existing line (multi-) graph with edge segment overlap. Figure 3.2 gives an example. We hence consider the following problem:

Problem 3 (Free Line Graph Construction)**.** *Given an arbitrary line (multi-) graph* $G = (V, E, L, \mathcal{L})$ *in which each edge is outfitted with a polyline describing its course. Construct a* free *line graph* G^* *from* G *which is suitable for rendering. In a free line graph, similar*

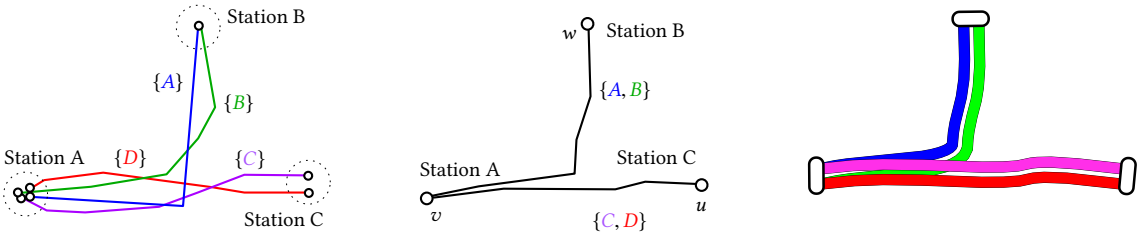


Figure 3.1: Left: Degenerate line graph in which edges corresponds to unique trips. Middle: Nearby nodes were first merged, and the edges in the resulting multigraph collapsed into single edges with averaged geometries. The resulting line graph does not track shared segments. Right: Map rendered from this line graph with line overlap.

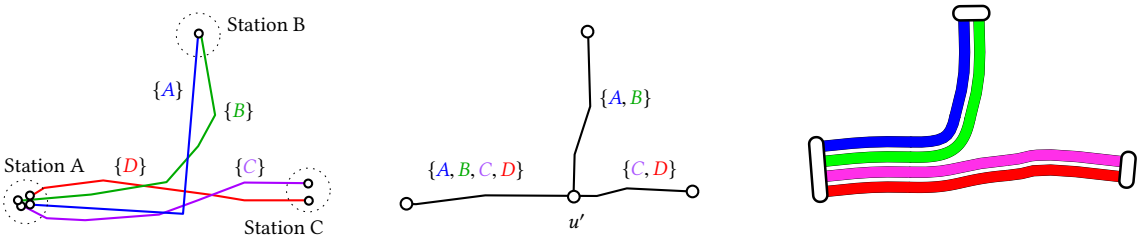


Figure 3.2: Left: Input line graph from Figure 3.1. Middle: Free line graph in which shared segments are represented by distinct edges. A new node u' marks the point where lines $\{A, B\}$ and $\{C, D\}$ branch. Right: Map rendered from this line graph.

stations should be represented by a single node (if possible), and edge segments overlapping (within some threshold distance $\hat{d}$) should be merged into a single new segment.

The remainder of this chapter is organized as follows: In Section 3.1, we discuss related work. In Section 3.2, we describe a simple and fast method to construct a *support graph* from the input graph G. Sections 3.3 and 3.4 then describe how we infer the original line turn restrictions, and the (clustered) stations back into the support graph. We give experimental results of our reference implementation in Section 3.5.

3.1 Related Work

Problem 3 can be considered a map construction problem, which is an active research area. There, the goal is to automatically derive the underlying road network from a collection of vehicle trajectories, typically GPS traces. One particular question that directly translates to our setting is how the quality of constructed maps can be assessed. There is also some overlap of our work with *edge bundling*.

3.1.1 Map Construction

As most mobile phones today are outfitted with GPS devices (or similar technologies such as Galileo, BeiDou, or GLONASS), a vast source of GPS traces exists. These may be used to construct maps for previously uncharted areas, augment maps constructed from aerial imagery, or update existing road network data. In the past, these task were typically performed manually. The advent of map services like Google Maps, Apple Maps, Bing Maps, or TomTom has led to an increased research interest in recent years, and hence several approaches to this problem have been developed. Ahmed et al. [1] group existing approaches into three categories: (1) point clustering based approaches, (2) approaches based on incremental construction, and (3) approaches based on intersection linking. These categories often blend, with many approaches using elements from multiple categories.

Point clustering based approaches assume a set of points as the primary input (they are typically obtained by using the raw input GPS points, or by sampling the input trajectories) and infer the map between these points (the raw GPS traces are often used to infer meta data, however). Already in 2003, Edelkamp and Schrödl [51] proposed to use the k-means algorithm to cluster the input points into k seed nodes. Two seed nodes are then connected if two points in the corresponding cluster were connected by an original trace (in this regard, it uses elements from intersection linking based approaches). The center line of the corresponding traces was used as an edge geometry. This approach was later refined in [130]. In particular, lanes and intersections were carefully reconstructed, and turn restrictions were considered by only connecting two reconstructed lanes if an original trace connected them.

Other approaches first transform the input traces into the black-and-white pixel domain and skeletonize the resulting image. For example, Davies et al. first compute a density grid [44] with each cell measuring the number of original input traces passing through it. Afterwards, this grid is transformed into the pixel domain and the contours of the edge segments are extracted. For the polygon edges, a Voronoi diagram is computed, and the nodes of the corresponding Voronoi graph that are outside the contour polygons are discarded. This results in a skeleton graph in which edges approximately follow the road contour centerlines, further cleaned up by removing very short edges. Input traces are then map-matched onto the skeleton graph to infer meta information.

Biagioni and Eriksson [28] use a similar approach, but smooth their 2D density grid of 1×1 meter cells via a convolution with a normal distribution density function. The resulting grid is then transformed into a black-and-white image using increasing thresholds, and a skeleton graph is successively constructed. The original traces are then also map-matched onto the skeleton graph, using a hidden Markov model. In this last step, unused edges are discarded, and turn restrictions are inferred. For overviews over point-clustering based map construction algorithms, see for example [27] or [1].

Incremental approaches begin with an empty map and iteratively add traces, typically merging new traces with existing segments. The final map is gradually constructed in this fashion. An early example of this approach was presented by Rogers et al. in 1999 [127] (their approach always augmented an existing map): given some existing map and a GPS trace t, sample both the map and the trace every 10 meters, merge each input point with the nearest existing map sample point, and average the node position.

A similar approach was used by Cao and Krumm [35], but they first mitigated GPS noise by applying a force-based pre-processing step on the GPS traces: each GPS trace point is attracted to its initial position, and to other trace segments, with a quadratically decreasing force. This bundles traces belonging to similar road segments closer together. Afterwards, the final map is constructed in a similar fashion as in [127], but if no merge node within a threshold distance is found, a new node is inserted. Our own approach for constructing the support graph is heavily inspired by the work of Cao and Krumm. Although we do not use a force-based approach for preprocessing, we first run our construction process with a very low threshold value to group nearby edge segments. Our construction algorithm then proceeds similarly to their work, but employs some additional heuristics to prevent the merge of map segments meeting at large angles, and the clear the map of merge artifacts.

Ahmed and Wenk [2] describe an incremental algorithm that is based on partial map-matching: using a previously described map-matching approach [5], they first find in a graph G embedded in $\mathbb{R}^2$ (modeling the map constructed so far) and a single polygonal curve C (modeling the GPS trace) the matching graph segments that optimize the overall Fréchet distance between the segments and C. For the partial map-matching part, the free-space diagram [6] corresponding to the maximum allowed Fréchet distance is established for each edge in G and C. These free-space diagrams are then glued together according to the edge adjacencies, and a monotone path through the resulting free-space surface that minimizes the usage of illegal (going through parts violating the Fréchet distance threshold) parts of the free-space surface is computed. The parts of the monotone path that pass through illegal parts are the unmatched segments for C. For these, corresponding new graph edges are added. The geometries of the matched segments are updated.

Intersection linking algorithms first establish the intersection nodes in the corresponding road network, and subsequently connect these nodes. Intersection nodes are generally defined as points in the network where vehicles often change their direction. As mentioned above, the k-means clustering based method described by Edelkamp and Schrödl [51] can also be considered an intersection linking method.

Fathi and Krumm [56] propose a method based on machine learning. They first define a local shape descriptor which is a circular window partitioned into annular segments. This shape descriptor then samples the input traces and for each sample point counts for each annular segment how many nearby traces are passing through it. The resulting cir-

cular histogram is then transformed into a feature vector. A classifier is trained to detect whether a local environment described by a shape descriptor belongs to an intersection. Given then a set of points that were classified as intersection nodes, they discard an intersection node if there was another intersection node with higher confidence within a threshold radius. The final road segments between intersection nodes are reconstructed by using the geometry of the shortest connecting trace.

Karagiorgou and Pfoser also consider the speed information contained in the GPS trace data [90] to extract intersections. Each GPS sample point is first checked whether it satisfies the turn condition. The latter is satisfied if the speed falls below 40 km/h, and if the vehicle direction changes by at least 15°. These so-called *turn samples* are then clustered by their proximity into intersection nodes, using a threshold distance of 25 meters. The intersection nodes are then connected by averaging all trajectory segments passing through them.

Some works focus exclusively on finding the road intersections. For example, Xie et al. [154] describe a method for finding intersections by searching for the Longest Common Subsequences (LCSS) of each pairwise GPS trace, using the start and endpoints of a subsequence as intersection points. These are then further clustered using Kernel Density Estimation.

Our own approach described in this chapter can be considered an incremental insertion algorithm. The main difference between our work and previous works are: (1) Our input data is typically not a collection of traces, but may already be a line (multi-) graph with many overlapping edge segments. (2) Our constructed line graph should contain *line information*, that is, for each edge e, we must establish the set $L(e)$ of lines traversing through e. (3) We must cluster station nodes, and re-insert these clustered stations at appropriate positions. (4) We do not aim at an exact geometrical reconstruction of intersection lanes, but want to specifically merge large intersections into single nodes, as this allows for a cleaner rendering of the map later on. Nevertheless, (line) turn restrictions at these intersection nodes should be considered. (5) We do not have access to speed information at individual sample points.

Before settling for the approach described in this chapter we experimented with other techniques. In [19] and [20], we presented an approach where we swept each input edge e and checked for other input edges within a threshold distance $\hat{d}$. As soon as such an edge f was found, we projected the position on e to f and marked the beginning of a segment. This segment was then extended along e and f as long as the distance between them was lower than d (we allowed for short outliers). The matching segment of e and f were then averaged, and nodes inserted at the beginning of the segment. Using various heuristics and clean-up rules, we tried to both track the lines traveling through the edges, the station positions, and the turn restrictions simultaneously. In practice, this approach lead to many artifacts which had to be removed with subsequent heuristic clean-up rules. It proved to be particularly difficult to correctly track turn restrictions

and to ensure that *similar* stations which were within the snap threshold were merged into a single station node in the final line graph.

3.1.2 Quality Measures

Early work on map-construction typically conducted a manual inspection of the constructed maps due to lack of machine-readable ground truth data. Over the years, several quality measures for comparing a map-constructed graph to a ground truth graph were developed. This problem is closely related to the general graph-theoretical problem of measuring the similarity between two graphs [42, 69, 65, 155], although spatial graph embeddings are typically not considered there [1]. Hence, several distance measures between map-constructed graphs and a ground truth graph have been developed. These range from simple approaches which densely sample both the constructed and the ground truth graph and compare the resulting point sets (e.g. by measuring the Hausdorff distance [7] or by computing specially developed measures [27, 103]) to path-based measures which compare selected paths through both graphs. For the latter, a random set of corresponding shortest paths is often used [108, 90].

3.1.3 Edge Bundling

There is also some overlap with our work and *edge bundling*, where the goal is to merge edges to decrease visual clutter when drawing complex graphs. In contrast to our work, where the original geographical line courses should be maintained, the edge embeddings are typically not part of the input and can be freely chosen to create as many bundles as possibles, while adhering to some optional side constraints. For example, Holten and van Wijk [83] describe a force-directed approach in which edges attract each other to form bundles. Pupyrev et al. consider edge bundling to produce graph drawings that look like metro maps, also without considering input embeddings. They first find an edge-bundled embedding which optimizes the total edge lengths and bundle widths. Afterwards, they optimize the edge orderings on the shared segments using a restricted variant of the metro-line crossing minimization problem (this will be discussed in Chapter 4).

3.2 Support Graph Construction

We first describe how to build a *support graph*. A support graph is a line graph in which all nodes are non-stations, and no turn restrictions are present. We build this support graph incrementally: given some input line graph G, we order the edges by the length of their geometries. These ordered edges $e_1, \ldots, e_n$ are then iteratively inserted into an initially empty support graph H_0, where H_i is the support graph after the insertion of

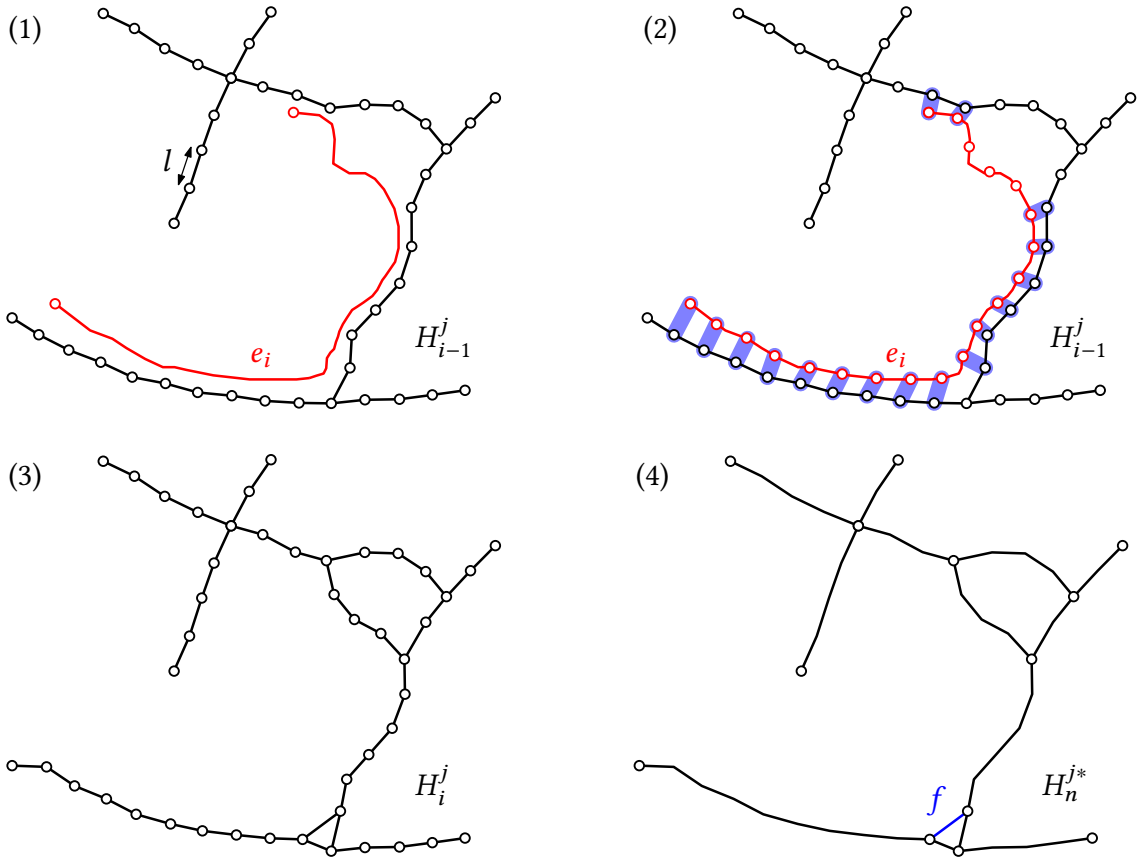


Figure 3.3: Constructing the support graph H^j for an input graph G. The edges e_i of G are iteratively inserted into H^j_{i-1} by first sampling them and merging sample points with the nearest existing node. Afterwards, degree 2 nodes are contracted where possible.

edge e_i. For the insertion of edge e_i, we densely sample the geometry of e_i and then merge sample points with the nearest existing nodes, if there are any within a threshold distance. An additional step, artifact cleanup, will be described in the following section. After all edges have been inserted into H, we start the process again with H as the input, and continue this until a convergence criterion is met. The support graphs constructed in reach round will be denoted as $H^1, \dots, H^m$.

Figure 3.3 gives an overview over the basic approach. We have an already existing support graph H^j_{i-1} from a previous iteration and want to insert edge e_i. The edge geometry of e_i is first sampled every l length units resulting in a list $p_1, \dots, p_l$ of coordinates. We then iterate over the coordinates and retrieve for each p_k the nearest existing node v in H^j_{i-1} from a grid index. If the distance between v and p_k is smaller than or equal to a threshold parameter $\hat{d}$, we average the position between v_k and p_k, move v to the resulting new coordinate, and set $v_k = v$. If the distance between v and p_k is greater than $\hat{d}$, we add a new node v_k for p_k. In both cases, an edge $f = \{v_{k-1}, v_k\}$ is added

connecting the node of the previous sampling point to v_k. We set $L(f) = L(e)$, that is f receives exactly the lines traveling on the original edge e. In case the edge $f = \{v_{k-1}, v_k\}$ already existed, we set $L(e) = L(e) \cup L(f)$. It is important to highlight that the edges in the intermediate support graphs H_i^j are not labeled with a polyline - their geometry is completely defined by the node embedding.

Observe that a naive application of this process would subsample the geometry of each input edge as we would always merge a sampling point p_i with the node of the previously inserted node as long as the distance is below $\hat{d}$. To avoid this, we keep a blocking set B of nodes corresponding to the last $\hat{d}/l$ sampling points and never use a node $v \in B$ as a candidate node.

After all edges has been inserted, we contract degree 2 nodes of the resulting graph H_n^j if (1) the adjacent lines have exactly the same lines, and (2) the contraction would not lead to a multigraph. During this process, the edges are re-labeled with polyline geometries: the geometries of the adjacent edges g and h are concatenated and used as the geometry for combined edge gh. This graph H_n^{j*} is then used as the input for a subsequent iteration, or returned if the convergence criterion is met.

A major limitation of this approach is that matching segments of H_{i-1}^j and e_i may not be completely merged. For example, in Figure 3.3, the blue edge f is left as an artifact. The next section describes how to handle such issues.

3.2.1 Artifacts

The process described so far leaves two types of artifacts, both depicted in Figure 3.4. Type 1 artifacts will be cleared by further iterations of our approach. A typical example is the triangular structure depicted in Figure 3.4, top. Node u was not contracted at the end of the first iteration as it would have led to a multigraph. After node u was added to the graph, another merge moved node v so that the u is now within distance $d < \hat{d}$ to e. If the length of e is above the sampling length l, a sampling point will be inserted somewhere along e in the next iteration, at which point the edges adjacent to u will be merged with the original edge e.

Type 2 artifacts typically involve one or more edges which are *shorter* than the sampling length l and adjacent to nodes of degree > 2. Figure 3.4, bottom gives an example. Here, edge e has a length below the sampling length l, which means that in a subsequent iteration, it will receive exactly 2 sampling points corresponding to the incidents nodes u and v. As e is shorter than the edges f and g, it will be inserted first into the support graph of the next iteration. At this point, the sampling point of node v will be snapped to the existing node at the same position, added during the construction of the blue edges. Node u will be placed at exactly the same position, as edges f and g are not yet inserted, and the sampling point corresponding to u cannot be snapped to the node corresponding to v, as it is in the blocking set B. Edge e will therefore be reconstructed

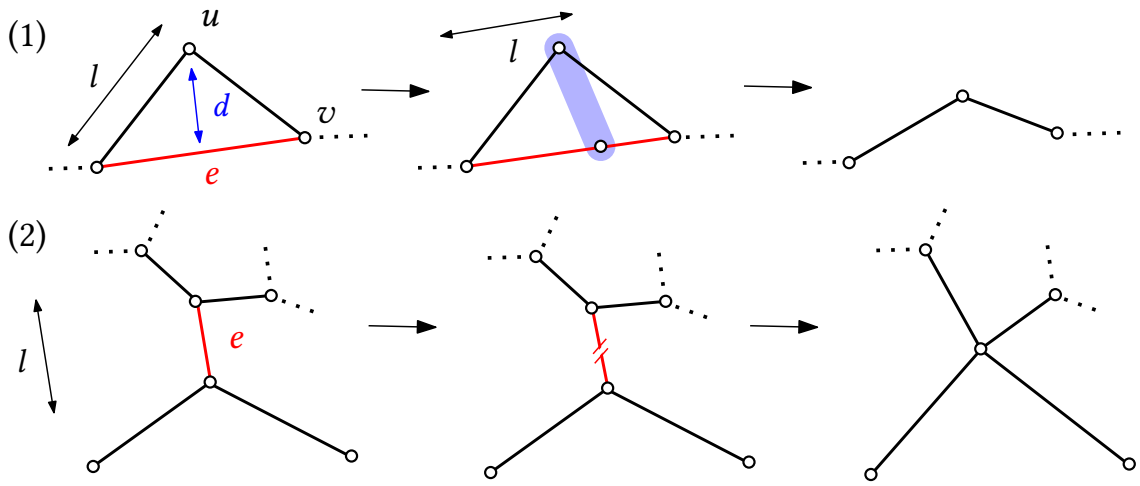


Figure 3.4: Two types of artifacts left by a single iteration of our approach. Top: If triangular structures remain in which the edges are within the merge threshold ($d \leq \hat{d}$), and if an affected edge e has a length greater than the sampling length l, an additional sampling node will be introduced in the next iteration, and the structure will collapse. Bottom: The edge e is shorter than the sampling length and adjacent to nodes of degree > 2. It will be reproduced exactly by following iterations of our approach. We immediately contract such edges after each iteration in a cleanup step.

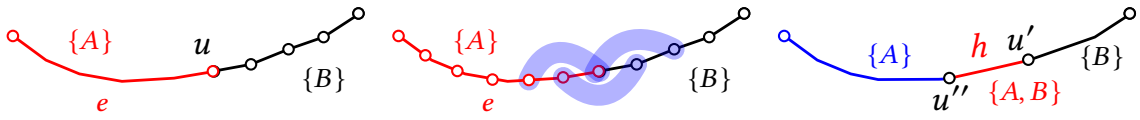


Figure 3.5: Left: The input graph consisted of two edges e and f meeting at an obtuse angle, f has already been inserted, e is to be inserted. Middle: the input edges will be interlaced during the support graph construction. Right: after degree-2 node contraction, a segment with lines A and B appeared. This *line creep* will continue with each iteration.

exactly. But if the length of e was below the merge threshold $\hat{d}$, we have missed a merge opportunity, and we will miss it in each subsequent iteration.

To avoid such issues, we contract edges adjacent to two nodes with a degree > 2 and which have a length below l immediately after the insertion of an edge e_i during the construction process. As two or more edges may form a single edge with length < l after the final degree 2 node contraction, we additionally clear such edges in the iteration's final degree-2-contracted graph H_n^{j*}.

Figure 3.6: Left: To solve the line creep issue shown in Figure 3.5, we define a no-merge zone for sampling points close to the first sampling point p_1, and the last sampling point p_l of e: if for any other sampling point p_i the distance to p_1 or p_l is smaller than to some merge candidate, we discard that candidate. Right: The resulting support graph.

3.2.2 Line Creep and Network Planarity Issues

A more subtle problem with the approach above is shown in Figure 3.5. Let e and f be two adjacent edges in the input graph G, with $L(e) = \{A\}$, and $L(f) = \{B\}$, whose geometries meet at an angle close to 180°.

Figure 3.5, left, shows a support graph into which f has already been inserted, and edge e should now be inserted. If e is now sampled as described above and the resulting sampling points snapped to H from left to right, the first sampling point of e which is in reach of the existing node u will be merged with u, and the subsequent sampling points will be merged with nodes corresponding to sampling points on the original edge f *after* u (again from left to right). Figure 3.5, middle, highlights the merged sampling points and existing nodes in light blue. This interlacing will lead to edges between the merged nodes to share the lines of e and f, resulting in a segment in which both lines are present (Figure 3.5, right). This will continue with each additional iteration until the original input edges e and f are merged into a single edge h with $L(h) = L(e) \cup L(f)$. We call this effect *line creep*.

To mitigate this issue, we add another rule to the node candidate selection process for the sample points: let $e = \{v, w\}$ again be the edge to be inserted into a support graph H, and let p_1 and p_l be again the first and last sampling point of e (corresponding to the position of v and w). Let now p_i with $1 < i < l$ be a sample point to be merged into H. If a node v in H at position p_v is within the merge radius of p_i, we only merge if $\alpha \cdot \text{dist}(p_i, p_1) > \text{dist}(p_i, p_v)$ and $\alpha \cdot \text{dist}(p_i, p_l) > \text{dist}(p_i, p_v)$($p_i$ is - within a factor α - nearer to v then to either p_1 or p_l). This effectively creates no-merge zones for sample points close to p_1 and p_l, as shown in Figure 3.6, left. We set α to $\sin(\pi/4) = 1/\sqrt{2}$. The intuition behind this is simple: if we assume that e.g. p_1, p_i and p_v form a right triangle, with the hypotenuse $\overleftrightarrow{p_1, p_i}$, then the condition is met as soon as $\overleftrightarrow{p_i, p_1}$ and $\overleftrightarrow{p_i, p_v}$ meet at an angle of less than 45°. It is easy to see that this prevents the merge of adjacent edges which meet at an angle greater than or equal to approximately 45°.

Another problem is that our approach so far tends to create complicated merge patterns between edges which cross in the input graph G, and which also cross in the "correct" free line graph we are trying to construct. Figure 3.7 gives an example. The input

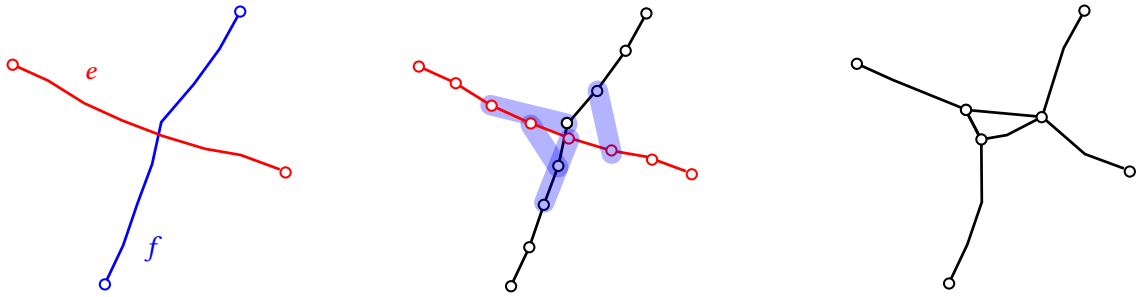


Figure 3.7: Left: Nonplanar input graph G where two edges e and f meet at an approximately 90° angle. Middle, right: during the support graph construction, an artifact merge zone between e and f will appear.

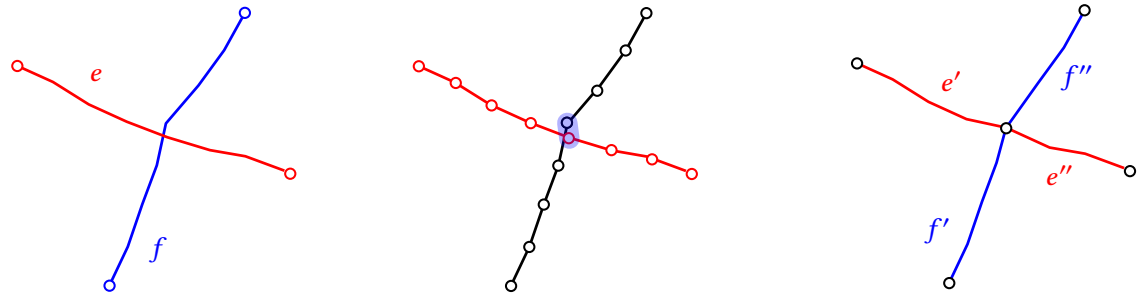


Figure 3.8: To avoid the merge zone artifacts shown in Figure 3.7, we first build a support graph using a very low merge threshold level ($2l$), which will planarize such intersection. The no-merge zones will prevent the edges e', e'', f', and f'' from being merged if the resulting support graph H is used as input for a construction with higher $\hat{d}$.

graph is not planar, and it is easy to see that there is no reason to merge the crossing edges e and f, as they should also cross in the final rendered map. As the edges cross in the middle, the no-merge zones described in the previous paragraph do not apply.

We avoid this with a simple preprocessing step: before constructing the support graph for G with merge threshold $\hat{d}$, we first construct a support graph H with a very small $\hat{d}$ (in our experiments, we used $\hat{d} = 2l$). This effectively planarizes the input graph, as shown in Figure 3.8. If H is now used as an input for a support graph construction with threshold $\hat{d}$, the original edges e and f have been broken up, and the resulting edges e', e'', f', and f'' meet at a common intersection node. At this node, the no-merge zones prevent a further merge if e and f crossed at an angle greater than 45°.

3.2.3 Convergence Criterion

As mentioned above, we apply the construction process in rounds, taking the final support graph H_n^{j-1*} as input for the construction of a support graph H_n^{j*}. Several conver-

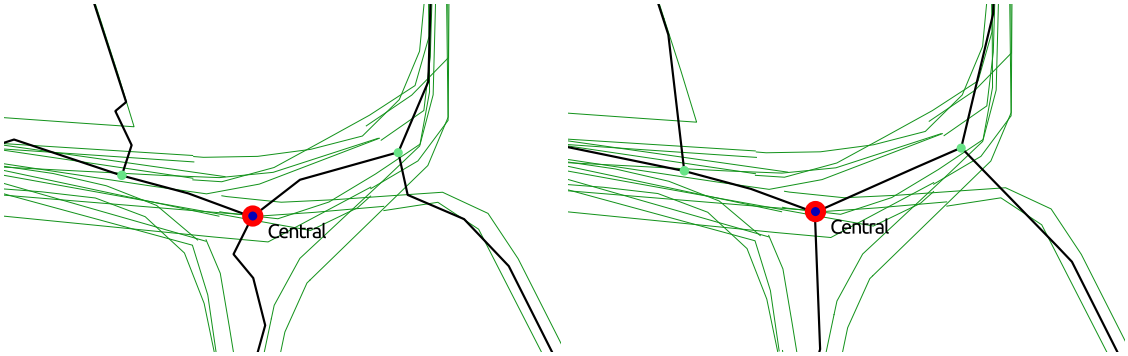


Figure 3.9: Part of the input line graph for the Zurich tram network (green), and the constructed free line graph (black). Edge geometries tend to be unsteady around large intersections. We remove the parts within a distance of $\hat{d}$ around the stations and replace it by a straight line.

gence criteria come to mind for this process, for example the number of lines, the number of edges, or the number of performed merges. During our experiments, we found the *edge length gap* $|1 - \frac{L^*}{L}|$ to work best. Here, L is the sum of all edge lengths in the input graph, and L^* is the sum of all edge lengths in the constructed support graph. In our experiments, we stopped the iterative process if the edge length gap fell below 0.2%.

3.2.4 Intersection Smoothing

Despite the artifact cleanup measures described above, the support graph construction process sometimes tends to slightly distort segment geometries at intersection nodes (see Figure 3.9, left for an example). To later achieve a clean look in the final drawing, we apply a final cleanup step: at each node v in the final support graph, we crop $\hat{d}$ length units of the geometry of each adjacent edge, set the position of v to the average of the resulting end points of the adjacent edges, and connect each edge with a straight line geometry to v. Figure 3.9, right gives an example of this process.

3.3 Inferring Line Turn Restrictions

Section 1.3.2 already introduced *line turn restrictions* in line graphs, but so far we didn't motivate them in detail. As mentioned in the introduction, line turn restrictions are important if lines don't follow simple paths through the network. Consider Figure 3.10, which shows our running example of the Chicago loop rendered with considering line turn restrictions (left), and without (right). Without them, the map shows several impossible turns for the blue, the brown, the light pink, and the orange line.

Previous work tried to track line turn restrictions through the support graph con-

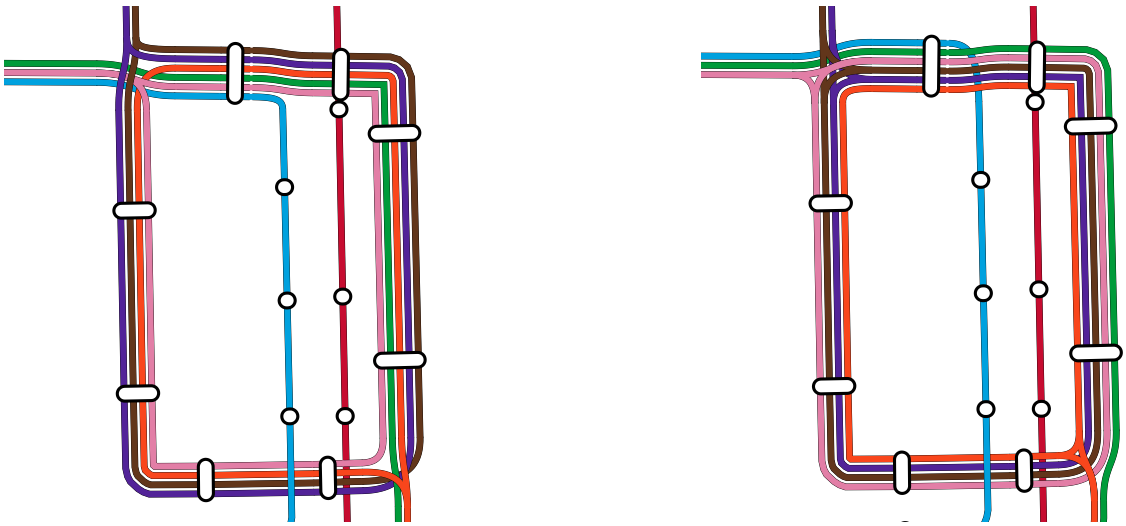


Figure 3.10: Left: Chicago loop, rendered with line turn restrictions considered. Right: Same map, rendered *without* considering line turn restrictions.

struction process [19], but this turned out to be very error-prone. In this section, we describe an alternative approach - instead of tracking the restrictions through the construction process, we infer them onto the final support graph. At first glance, it might seem enough to track the original edges merged into support graph edges during the construction process and determine the turn restrictions based solely on edge adjacencies. For an edge h in a support graph H, let $M(h)$ be the set of original input edges merged into h during the construction of H. If then, for example, input edges e and f were adjacent in G, and we now check two edges h and i with $e \in M(h)$ and $f \in M(i)$, then if some line l continued from e to f in the original input graph, we would determine that l also continues from h to i. However, depending on the merge order of input edges, overlapping input edges may lead to an input edge e appearing in $M(h)$ although it only shared a short segment (see Figure 3.11 for an example of how that would distort line turn restriction based on the adjacencies of the $M(e)$). We use a more robust method of comparing the lengths of shortest paths connecting special *edge handle* nodes in the support graph to the shortest path lengths between corresponding handles in the original input graph.

3.3.1 Edge Handles and Shortest Path Length Comparison

Consider Figure 3.12. Given an support graph H constructed from G, we first add edge handles to each edge in H. These are two nodes inserted at a progression of $1/3$ and $2/3$ on the corresponding edge geometry (Figure 3.12.2). This results in a support graph H' with edge handle nodes. For each pair of original support graph edges e and f adjacent at a node u, we now have a pair of handle nodes α, β, where α is the handle node on e

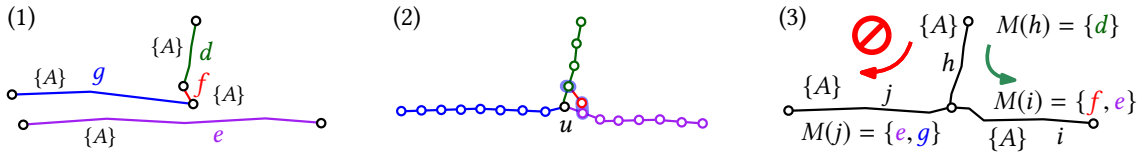


Figure 3.11: Turn restrictions based on adjacencies of merged edges distorted during the support graph construction. (1) Input graph G. (2) After edges e, g, d, and f (in that order) have been inserted into H, edges stemming from d and g are now adjacent at node u. The next iteration will connect the edge stemming from f with an edge stemming from e. (3) After degree-2 node contraction, the adjacencies of the merged original edges $M(h)$, $M(i)$, and $M(j)$ incorrectly infer a line turn restriction from h to j for line M, and no line turn restriction from h to i.

which is closest to u, and β is the handle node on f which is closest to u.

In a second step, we draw an imaginary line (orthogonal to the corresponding support graph edge) of length $a\hat{d}$ (our merge threshold times some padding factor a) through and centered at each handle node and add corresponding handle nodes to the original input graph where edges intersect with the line (Figures 3.12.2 and 3). The padding factor a is required as the distance between a handle point on e and one of the original edges merged into e might be greater than $\hat{d}$, depending on the merge order. This results in a line graph G' with edge handle nodes. We now have for each handle node α in the support graph H' a corresponding set of handle nodes $\mathcal{A}$ in the original input graph G.

3.3.2 Shortest Path Comparison

For each pair α, β of edge handles in H', we compute the shortest path from α to β through the support graph H' (with edge weights based on the edge lengths), and store its cost. We then also compute the cost of the set-to-set shortest path from $\mathcal{A}$ to $\mathcal{B}$ through G', but this time separately for each line $l \in L(e) \cap L(f)$. In particular, we set the weight of an edge e in G' to ∞ if $l \notin L(e)$. Importantly, we now also consider turn restrictions present in the original input line graph G, using the same approach as described in Section 2.6.4 (an edge-to-node dual graph). If for two edges e and f (adjacent at u) and some line l, the difference between the shortest path cost from their edge handles α, β through H' and the shortest path cost under l from $\mathcal{A}$ to $\mathcal{B}$ through G' is greater than a threshold t, we add a turn restriction from e to f at u for l. In our experiments, we used a threshold t of 500 meters.

For example, in Figure 3.12.4, the shortest path through G' from $\mathcal{A}$ to $\mathcal{B}$ for the red line l would take the entire loop, while the shortest path from α to β in H' would be much shorter and go through node u.

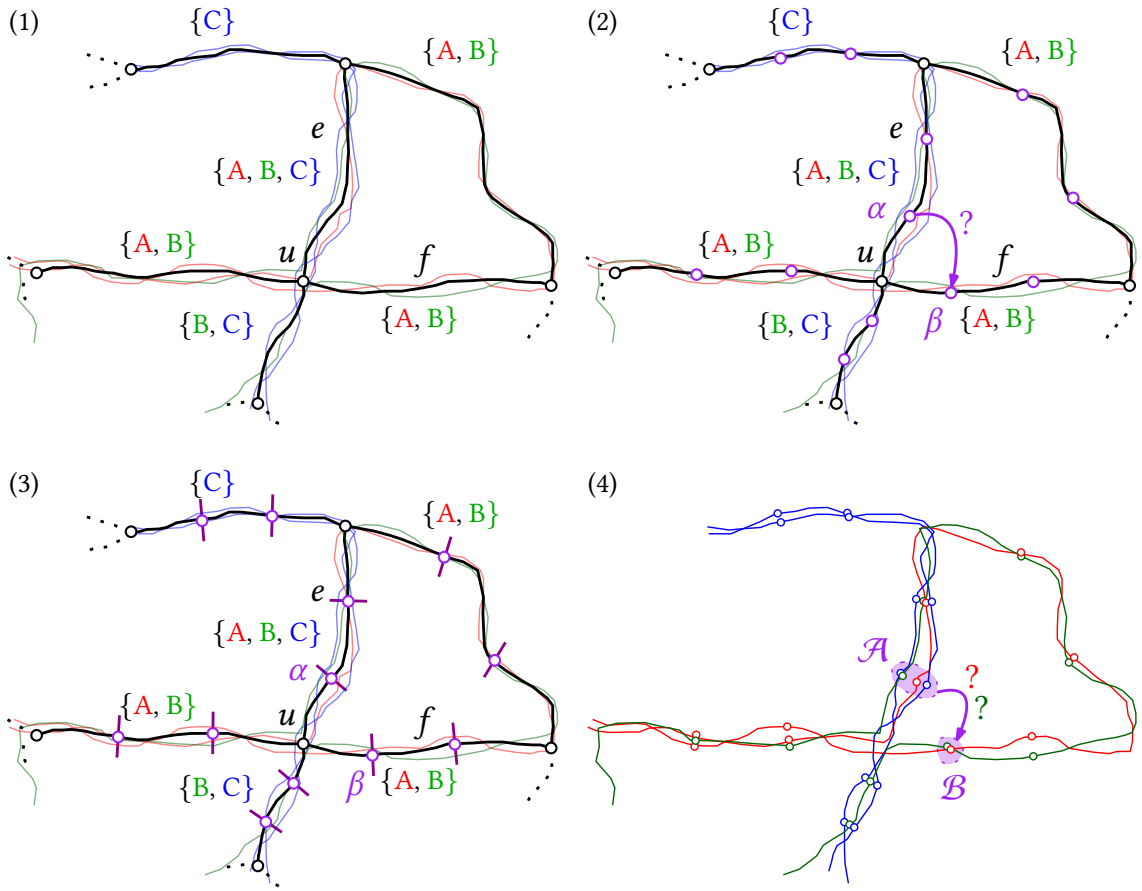


Figure 3.12: Inferring turn restrictions onto a support graph H. Given a support graph H for an original input graph G (1), we first construct a support graph H' in which for each edge two handle nodes are inserted at a progression of 1/3 and 2/3 of the edge geometry. An imaginary line is then drawn through edge handle nodes, orthogonal to the corresponding edge, and a set of handle nodes is added to the edges in G intersecting this imaginary line. For two edges e and f adjacent at u, we compare the shortest path connecting the handle points closest to u with the shortest path from the corresponding set of handle nodes added to G, for each line l. If the shortest path cost difference is above a threshold t, we add a turn restriction for l from e to f at u.

3.3.3 Avoiding False Positives

Depending on the size of the padding factor a and the position of the imaginary cutting line, we might add a handle point for an edge e in H to an edge f in G which wasn't merged into e during the support graph construction process. To avoid such false positives, and to also speed up the selection of appropriate edges, we maintain for each edge e in the support graph H a set $M(e)$ of original input edges that have been merged into e, just like we keep track of the lines on e through the construction process. If the imaginary cutting for a handle point on support graph edge e line intersects an edge f in the input graph, but $f \notin M(e)$, we do not add an edge handle point there.

3.4 Station Clustering and Insertion

It now remains to show how the original stations can be re-inserted into the support graph in a clustered way. We do this by selecting candidate insertion edges for each station cluster, ordering them by a score, and inserting a node into the support graph at the best candidate position.

Let S be the set of station nodes in the original input line graph G, and let each station node $s \in S$ be labeled by a function n which assigns s a station label (for example, $n(s) =$ Main Station). Observe that each s is served by a set of edges in G, namely $\text{adj}(s)$, and recall that we have kept a set $M(e)$ for each edge e in the support graph H which holds all original input edges merged into e during the support graph construction. As we desire close similar stations to be represented by a single node in the final map, we first partition S into subsets of *similar* stations. Let P be this partition. To decide whether some $s \in S$ belongs to a subset $A \in P$, we use a $P + JAC$ similarity classifier as described in Section 2.8 and put s into A if $\exists a \in A : \text{sim}_{\text{JAC+P}}(s, a) > 0.5$. Note that this implies that no two subsets $a, b \in A$ share a similar station. Additionally, we keep for each station cluster A a set Q_A of edges served by the $s \in A$, that is $Q_A = \bigcup_{a \in A} \text{adj}(a)$.

For each A, we then retrieve a set of edges in H that are within a threshold distance r around the centroid of the $a \in A$. For each such retrieved edge e, we add 3 candidate positions to the candidate set: the centroid projected onto the geometry of e, and the position of the two adjacent nodes of e. Each candidate c then receives a score based on (1) the distance d from the centroid of A to c, and (2) the number C of *correctly served* edges at c. If c was the projected coordinate on e, then $C = |Q_A \cap M(e)|$. If c was a node v, then $C = |Q_A \cap \bigcup_{e \in \text{adj}(v)} M(e)|$. Based on these values, we calculate a score

$$o = d + \frac{C}{Q_A} \cdot 100 \tag{3.1}$$

and insert the station cluster at the candidate of highest score.

3.5 Experimental Evaluation

This section briefly discusses the experimental results as well as existing problems (and possible solutions) of our approach. We conducted a quantitative and a qualitative evaluation on 6 schedule datasets: a simple tram network with few necessary line turn restrictions (Freiburg-Tram), a medium light rail network with several necessary line turn restrictions in which segments often meet at rectilinear angles (Chicago, thus testing the effectiveness of our no-merge zones technique described above), a large and complicated subway network in which many line courses cross (New York), an equally complicated tram network requiring many line turn restrictions (Zurich-Tram), as well as medium-sized bus network (Freiburg-Bus) and a large bus network (Zurich-Bus). The Zurich datasets were also chosen because the official schedule dataset has imprecise shape data, leading to large edge bundles on each network segment (as shown in Figure 3.19).

3.5.1 Experimental Setup

Preliminary line graphs were first extracted from the schedule datasets for each of the test networks using a straightforward approach: each stop in the GTFS dataset yielded a node, and line connections between stops yielded a single edge, labeled with a geometry that was extracted from the GTFS shapes. If no shapes were available, we first map-matched the dataset using our approach from Chapter 2 (this was only necessary for the Freiburg dataset). From these preliminary line graphs, we then constructed a free line graph using the method described in this chapter. As a segment merge threshold, we used $\hat{d} = 50$m. For a smoother look, we also applied a single round of Chaikin smoothing to the final line graphs [36]. The preliminary line graphs extracted from the schedule data as well as the evaluation setup are available online[1].

For the quantitative evaluation, we used the preliminary line graphs as ground truth and compared them to the constructed final line graphs using a shortest path-based distance measure. We first densely sampled both the ground truth graph and the test graph. Then, 10,000 random node pairs (u, v) were picked from the ground truth graph. For both u and v, we then picked all nodes within a distance of $3\hat{d}$, for both the ground truth and test dataset. This yielded corresponding context sets U_G and V_G in the ground truth graph, and U_T and V_T in the test graph. Afterwards, we computed the shortest set-to-set path p_G connecting U_G and V_G through the ground truth graph, and the shortest set-to-set path p_T connecting U_T and V_T through the test graph, both for a random line $l \in \mathcal{L}$. The resulting paths were then compared using the Fréchet distance. This measured (1) the topological correctness of the constructed line graph edges (and the lines traveling on them), (2) the quality of the line graph edge geometries, and (3) the correctness of the inferred turn restrictions. To also measure the quality of the re-inserted stations, we

[1] `https://github.com/ad-freiburg/topo-eval`

Table 3.1: Quality of our line graph construction approach. We took 10,000 pairs of random sample nodes and calculated the shortest path connecting the corresponding context sets (all nodes within $3\hat{d}$ meters around the sample node in the input line graph and the constructed free line graph). The experiment was repeated for a set of random station pairs. Under *acc.* we give the percentage of paths with a Fréchet distance under $3\hat{d}$. Under δ_F *min*, *max*, and *avg.*, we give the minimum, maximum, and average Fréchet distance over all paths.

	Random sample pairs				Random station pairs			
	acc.	δ_F min	δ_F max	δ_F avg.	acc.	δ_F min	δ_F max	δ_F avg.
Freiburg-Tram	98%	0.6	282.4	23.1	90%	0.63	159.9	42.8
Chicago	99%	0.2	1481.3	26.9	88%	0.1	1410.3	24.6
New York	95%	0.1	2575.5	40.9	61%	0	18682.9	150.4
Zurich-Tram	81%	1.2	4441.7	68.6	78%	1.9	2660.1	104.1
Freiburg-Bus	74%	1.1	2881.1	73.3	56%	1.8	3783.5	133.3
Zurich-Bus	71%	3.8	1486.4	103.9	61%	2.8	4212.1	166.9

conducted a similar experiment where we did not use random sample nodes for the context sets, but random station pairs from the ground truth data. The context sets were then stations from the ground truth and the test data that were *similar*.

We also measured the time and number of iterations required until convergence, the time needed for inferring turn restrictions, and the time needed to re-insert the clustered stop positions.

3.5.2 Results and Discussion

The results of our quantitative evaluation are given in Tables 3.1 and 3.2. The resulting free line graphs for the Freiburg-Tram, Chicago, New York, and Zurich-Tram datasets are also shown at the end of this chapter in Figures 3.15, Figures 3.16, Figures 3.17, and Figures 3.18. Figure 3.19 additionally shows an excerpt of the Zurich city center area.

Table 3.1 gives the minimum, maximum, and average Fréchet distances between the shortest paths. We also computed an accuracy score, which is the percentage of paths with a Fréchet distance under $3\hat{d}$. Under this measure, and for random sample pairs, we found the quality for Freiburg-Tram, Chicago, and New-York to be near-perfect, all with an accuracy near 100%. For Zurich-Tram, the accuracy was 81%. Under manual investigation, the main culprit for this lower score was an incorrect line turn restriction near the station *Stauffacher*, which was caused by a loop-like edge geometry in the preliminary line graph extracted from the GTFS data. For the Freiburg and Zurich Bus network,

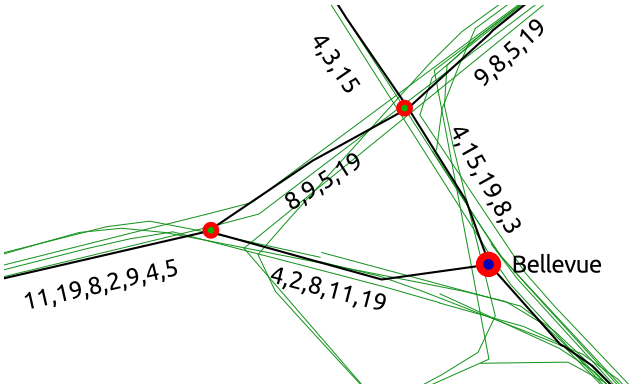


Figure 3.13: Excerpt from the free line graph of Zurich around the Bellevue station. A single station node was inserted for the clustered Bellevue station, which is not adjacent to line 9 which served Bellevue station in the input line graph, and a turn restriction prevents line 8 from serving the station. It is not possible to insert a single station in such a way that all lines serve the station.

the reasons for the lower accuracy scores of 74% and 71% were also often problems with the inferred turn restrictions, but there were also many cases in which the Fréchet distances between shortest paths was just slightly above $3\hat{d}$, suggesting that the resulting geometry was correct, but a series of edge merges has moved the final edge geometry more than $3\hat{d}$ from an original input edge geometry (this often happened around stations covering a large area which were iteratively merged into a single node, with edge geometries further distorted by the intersection smoothing described above).

For random station pairs, the accuracy scores were significantly lower. The main reason for this is that we are currently only inserting a *single* merged station node for a cluster of similar stations. For large stations, however, it is sometimes impossible to place a single node in such a way that all the edges (and therefore lines) originally serving the station cluster are still serving the merged station node in the final free line graph. Figure 3.13 gives the example of the Zurich Bellevue station, where line 9 does not serve the inserted station node in the final line graph, and turn restrictions prevent line 8 from serving it. Here, the station intersection should either be collapsed into a single degree-4 node, or an additional station node should be inserted.

To measure the general quality of the edge geometries, we did a careful manual investigation and noticed 5 kinds of common remaining problems over all datasets, as shown in Figure 3.14:

S1. Edge geometries are sometimes not smooth. Depending on the order in which nodes are merged, the resulting final polyline may appear unsteady.

S2. Depending on the merge order of edges, intersection nodes may be moved towards a certain segment.

S3. Intersection smoothing is at the moment also applied to nodes of degree 2. This sometimes removes geographical details within the merge threshold around such nodes (see the Wipkingerplatz station node in Figure 3.14).

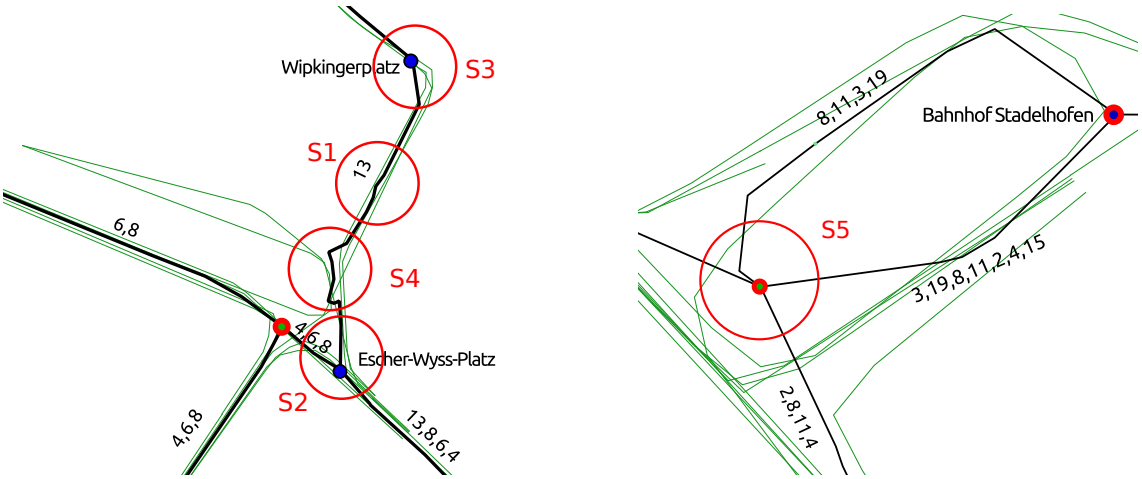


Figure 3.14: Excerpt from the Zurich tram line graph highlighting remaining problems of our approach: (S1) Unsteady edge geometries. (S2) Lost geographical details around degree 2 nodes. (S3) Displaced intersection nodes. (S4) Outlier trips may distort the line graph. (S5) Intersection smoothing might add new merge opportunities.

S4. Infrequent trips with extreme course deviations may distort the line graph. In the example, the outliers are vehicles returning to a depot at Escher-Wyss-Platz in the evening.

S5. Intersection smoothing might introduce additional merging opportunities.

Additional smoothing after each iteration, each merge step, or after the entire support graph construction might be able to mitigate problem 1. A root cause for it (and also for problem S2) is that the input edges so far carry no weight. If an edge e in the support graph is already the results of 5 merges, and we are now merging a sixth edge f onto it, then the new positions of the merged nodes will again be the average positions of the node belonging to e, and the node belonging to f. It might be better to weight the positions by the number of already merged edges.

Regarding problem S3, the obvious solution is to reconstruct intersections only for nodes of degree greater than 3. Problem S4 is again related to problems S2 and S3, but this problem suggests a weighting of the edges not by the number of original edges (and thus the unique trips with the same shape on this edge), but by the total number of trips traveling through this edge in the input schedule data (either in total, or on an average weekday). It might also be sensible to consider completely dropping line connections from the input data on which the total number of trips on an average weekday is below a certain threshold (for example below 10% of the average number of line connections in the entire dataset), as such trips are usually not depicted in official network maps.

The number of iterations until convergence as well as the running time, the number of inferred restrictions, the maximum number of input edges merged into a single edge, and the overall edge lengths in the input and free line graph are given in Table 3.2. In

Table 3.2: Under *it*, we give the total number of support graph construction iterations until convergence. $t_{|H|}$ is the time required for the support graph construction, t_R the time required for the inferring of line turn restrictions, and t_S the time required for the station insertion. The number of inferred line turn restrictions is given as $|R|$. K is the maximum number of input edges merged into a single free line graph edge, and l_{bef} and l_{aft} is the total summed edge length of the input graph (l_{bef}) and the constructed free line graph (l_{aft}).

	it	$t_{\|H\|}$	t_R	t_S	$\|R\|$	K	l_{bef}	l_{aft}
Freiburg-Tram	6	1.1s	9ms	7ms	4	27	115.9k	52.2k
Chicago	4	3.6s	30ms	155ms	5	22	490.1k	220.3k
New York	5	7.2s	80ms	57ms	12	29	679.1k	442.9k
Zurich-Tram	15	11.2s	50ms	47ms	95	78	345.9k	125.8k
Freiburg-Bus	7	10.9s	136ms	43ms	238	61	930.7k	294.4k
Zurich-Bus	15	1.7m	1.1s	0.5s	1k	64	6741.1k	2496.3k

general, our approach quickly converged in under 16 iterations for all test datasets, and a line graph was constructed in under 12 seconds for all datasets except the Zurich bus network (here, the summed length of the input geometries was nearly 6,700 kilometers). We are confident, however, that the running time can be further reduced with additional engineering effort, for example by restricting the number of merge candidates for each inserted sample point.

The total lengths of the final free line graphs also seem to closely match the official network lengths reported by the transit companies. For example, for the Freiburg tram network, the summed edge geometry lengths were 52.2 km, and the official network length is 43.9 km[2]. For the Zurich tram network, these lengths were 125.8 km, and 125.1, respectively[3]. For the bus networks, the deviations were significantly greater (294.4 km vs 173.1 km for Freiburg, 2496.3 km vs 408.8 km). However, we note that the official numbers were only for the single main urban bus company and the schedule data also contained many regional lines. The official numbers also usually do not count outlier lines or lines that use a slightly different course in different directions. These numbers were thus hardly comparable.

3.6 Conclusions and Future Work

This chapter presented a method to construct a *free* line graph which can be rendered into a transit map without edge overlaps from raw schedule data, or from a line graph in

[2] `https://de.wikipedia.org/wiki/Freiburger_Verkehrs_AG`
[3] `https://de.wikipedia.org/wiki/Verkehrsbetriebe_Z%C3%BCrich`

which such overlaps would be present (for example, a line graph directly extracted from OSM data), a task that is crucial for the final visual quality of the maps. To achieve this, we first constructed a *support graph* without any station nodes by iteratively inserting edges from the input data into the initially empty support graph, merging nodes within a threshold distance $\hat{d}$. We used several heuristics to stabilize this approach. We described a method to infer line turn restrictions present in the original input data onto this support graph, and described how the stations can be re-inserted into this support graph in a clustered manner. For the clustering process, we used the station similarity classifier from the previous chapter. We conducted an extensive quantitative and qualitative evaluation.

Although we still consider the results of this chapter preliminary, they already produced very good results, with remaining problems regarding the quality of individual segment shapes, and the placement of the clustered stations. For the former, we see great potential in weighting the sample points by the number of edges (or trips) already merged into the edge they were sampled from, and considering this weight when averaging the new position. For the latter, it might be necessary to insert more than a single node into the final free line graph for a station cluster.

It would also be interesting to evaluate our approach not on input data extracted from schedule datasets, but from geographical datasets like OSM.

We note that our methods are not restricted to the domain of line graphs for transit map rendering, but can in theory be applied to arbitrary graph-like input data. In particular, our method should also work for inferring turn restrictions onto road networks constructed from vehicle GPS traces.

Figure 3.15: Line graph for the 2022 tram network of Freiburg, created from raw schedule data using the approach described in this chapter. The input schedule data was first map-matched with our approach from Chapter 2. The merge threshold was set to $\hat{d} = 40\,\text{m}$. Stations are given in blue, non-station nodes in green, and nodes with a red outline have inferred turn restrictions. Not all station and line labels are shown.

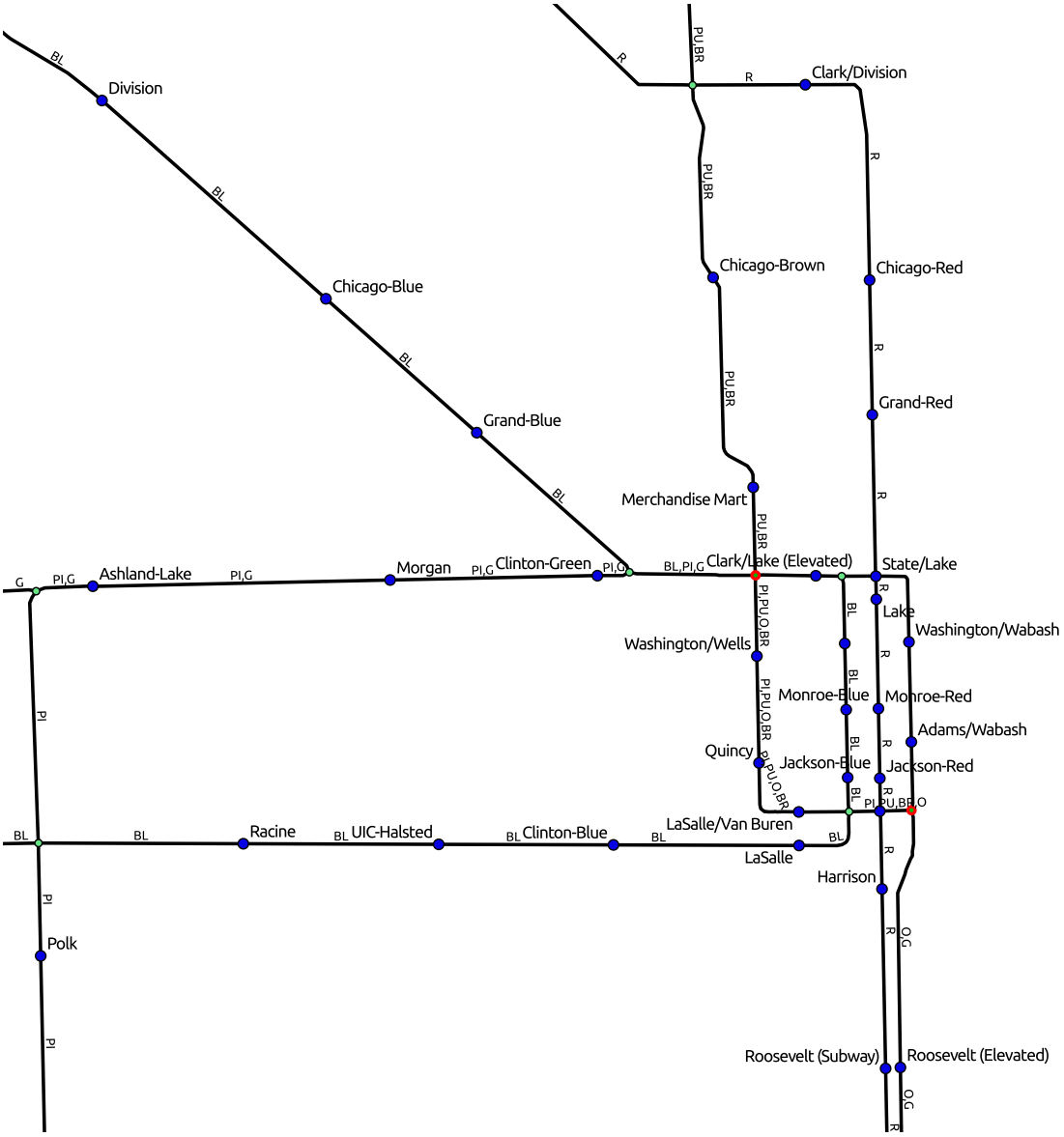


Figure 3.16: Excerpt from the line graph for the 2020 'L' network of Chicago, created from raw schedule data using the approach described in this chapter. The merge threshold was set to $\hat{d} = 40\,\text{m}$.

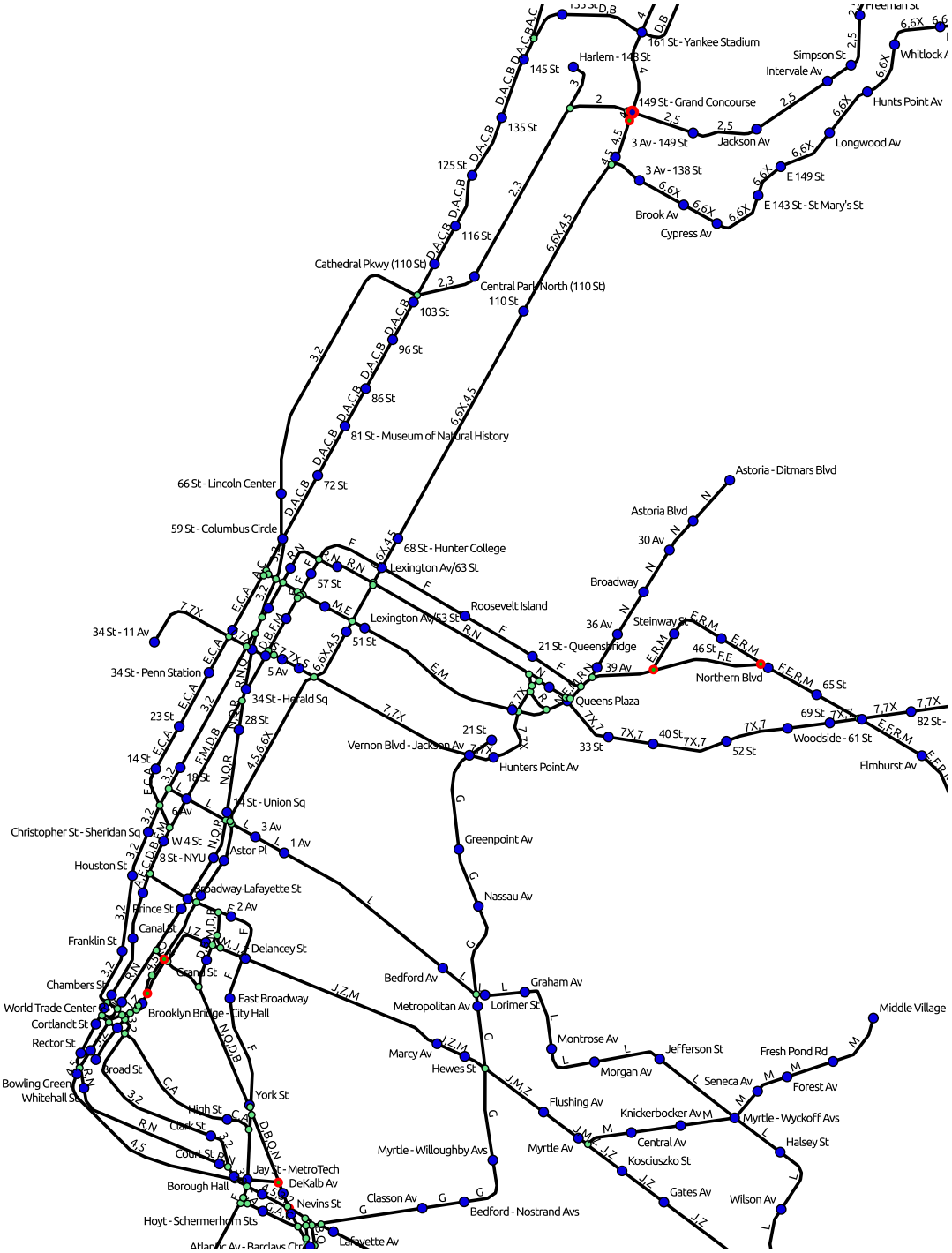


Figure 3.17: Excerpt from the line graph for the 2018 subway network of New York, created from raw schedule data using the approach described in this chapter. The merge threshold was set to $\hat{d} = 40\,\text{m}$.

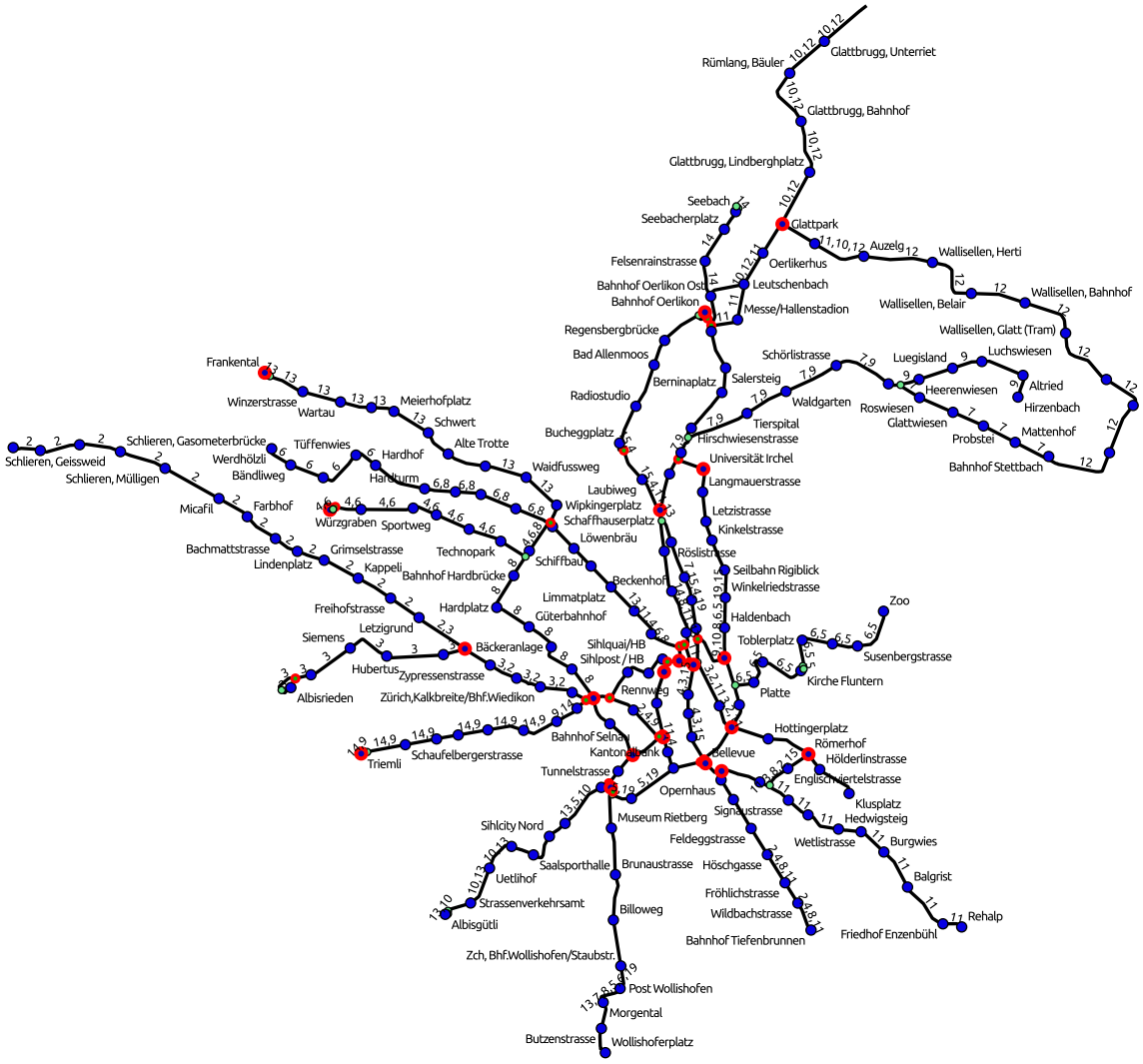


Figure 3.18: Line graph for the 2021 tram network of Zurich, created from raw schedule data using the approach described in this chapter. The merge threshold was set to $\hat{d} = 40\,\text{m}$.

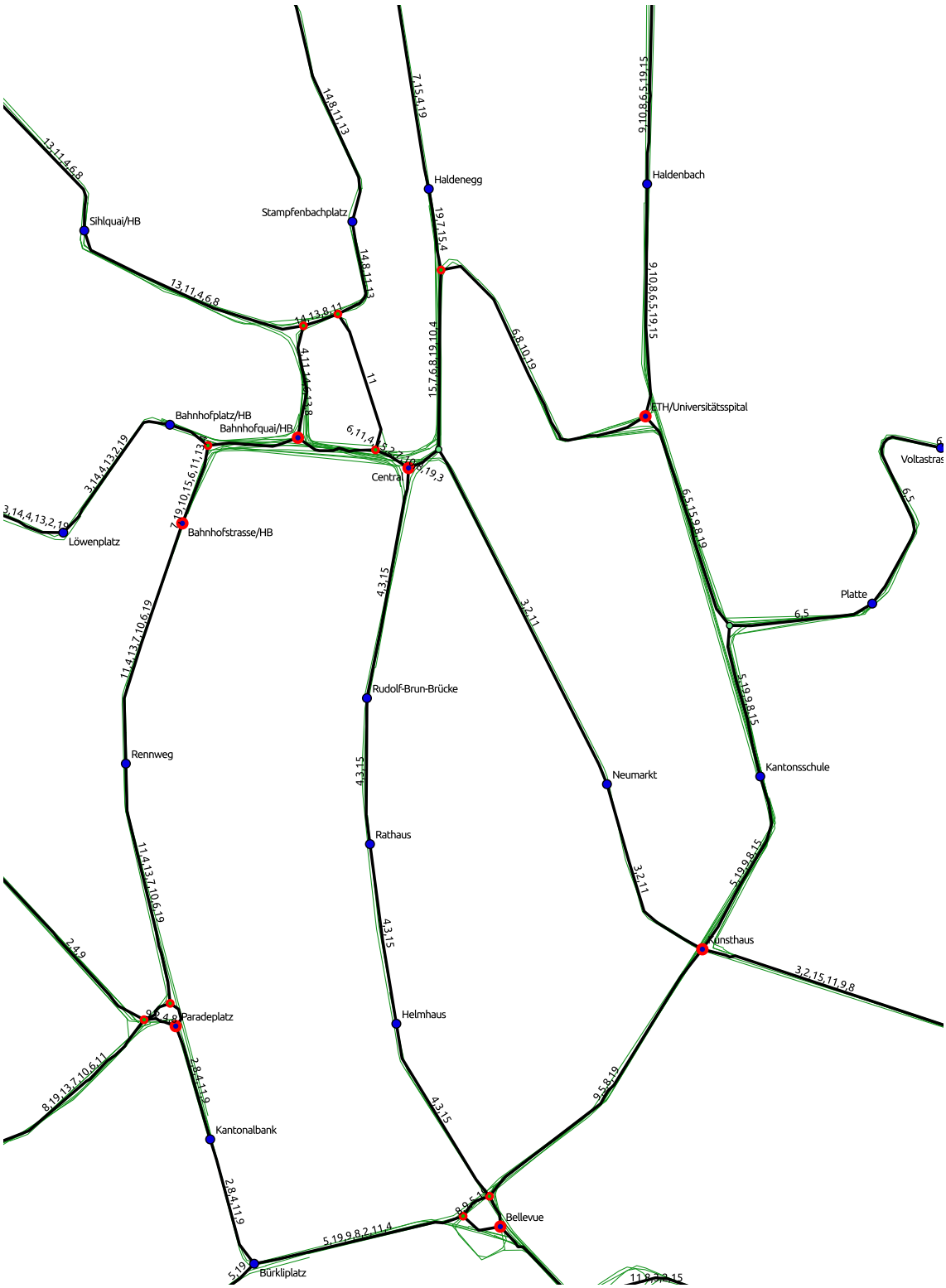


Figure 3.19: Excerpt from the Zurich line graph (black) given in Figure 3.18 around the city center, with the raw input data (green).

Chapter 4

Line Ordering Optimization

Our pipeline so far has produced an undirected line graph $G = (V, E, L, \mathcal{L})$, where for each $e \in E$, $L(e)$ labels the edge with a set of transit lines passing through e. While we may already render this line graph as a transit map, the result will be neither esthetically pleasing nor informative (see Figure 4.1 for an example). Because $L(e)$ is an unordered set, it is not clear in which order the lines on e should be rendered. If we just use a random ordering (or the input ordering) it will produce many situations where transit lines *cross* or *separate*, making it very hard to follow line courses.

This chapter formalizes the problem of finding permutations (we say a *line ordering*) of the $L(e)$ that are *informative*. We describe and evaluate several algorithmic approaches to find such an optimal ordering very fast. In particular, we would like to have as few undesired line crossings as possible and therefore consider the following problem:

Problem 4 (Optimal Line Permutation)**.** *Given a line graph $G = (V, E, L, \mathcal{L})$. For each edge $e \in E$, find a permutation of the $L(e)$ such that the readability of the final map is optimized. We call such a permutation a* line ordering*, and the set σ of all line orderings for E a* line ordering solution*.*

We will specify what constitutes optimal readability later on.

Our contributions in this chapter are the following: We give a formulation of the Metro Line Crossing Minimization problem (MLCM) [26], which we call the Metro Line Node Crossing Minimization problem (MLNCM). This formulation better matches real-world transit maps and also simplifies the map rendering process (a similar formulation has previously appeared in the context of wire routing for electronic designs [75] and edge bundling [123]). We then introduce a weighted version (MLNCM-W) which allows specifying line crossing weights per node and line pair. We also introduce the novel concept of minimizing *line separations* and formulate the MLNCM problem with counting line separations (MLNCM-S) and with counting *weighted* line separations (MLNCM-WS). MLNCM-WS will be shown to be able to produce line orderings very similar or

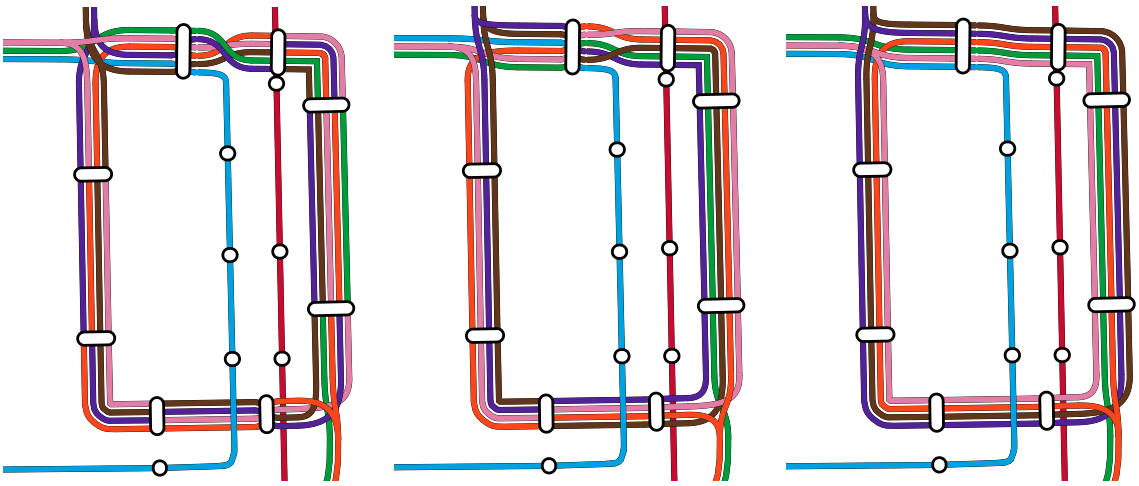


Figure 4.1: Left: The "Loop" of the Chicago light rail system with the unoptimized input line ordering; it is very hard to follow lines. Middle: Same area, with an imperfect line ordering (obtained via a local search approach). Right: Optimal line ordering.

equal to manually created line orderings in official maps. The concept of line separations partly replaces previous approaches to solve the problem of *block crossings* (this will be further explained below). We give NP-hardness proofs for MLNCM, MLNCM-W and MLNCM-WS (the NP-hardness of both MLNCM-W and MLNCM-WS directly follows from the NP-hardness of MLCM, however, in its raw MLNCM version, the problem seems to defy a straightforward reduction proof from MLCM). On the positive side, we describe a linear-time algorithm to solve MLNCM to optimality on line graphs where each line follows a simple path and ends at a node of degree 1. We also give a fast ILP formulation to optimize MLNCM-WS (and thus MLNCM, MLNCM-W, and MLNCM-S), orders of magnitude faster than a simple baseline formulation using state-of-the-art ILP solvers. Additionally, we describe and evaluate several heuristic approximation approaches to this problem. We also describe a set of pruning, cutting and untangling rules which may be applied to the input line graph to (sometimes dramatically) reduce the search space size, provably without affecting optimality. These *line graph simplification rules* can be fully applied in polynomial time and are independent of the actual method used to *find* the optimal ordering. Their attractiveness additionally stems from the fact that they always yield a set of MLNCM subproblems. These may then be solved in parallel using an appropriate method, which may be heuristically selected. We also describe cases where these simplification rules already produce an optimal line ordering solution, as the input graph decomposes into a set of simple components, each with a search space size of just 1. Finally, we conduct an extensive experimental evaluation of our methods on several real-world networks and demonstrate that they are fast enough to be used interactively, for example in a map editor. Most of the results of this chapter have been previously published in [19] and [20].

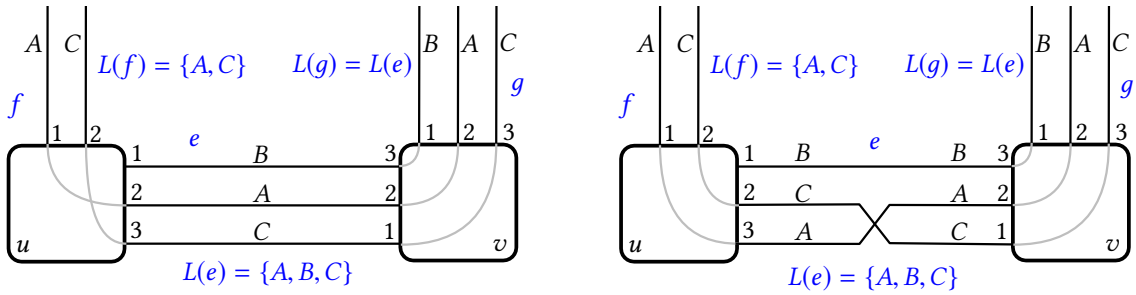


Figure 4.2: Left: In our MLNCM formulation, we require a *single* ordering p_e of lines on an edge $e = \{u, v\}$. This ordering gives the position of a line $l \in L(e)$ on e. This position may be stated either w.r.t. u (then denoted as p_e^u) or v (then denoted as p_e^v). Crossings occur at nodes (here, A and C cross at u). Right: In contrast, the original MLCM formulation allows for *two* orderings per edge e, which are not necessarily compatible (if they are not, a crossing occurs on e). Additionally, line orderings of adjacent edges must be compatible. For example, the crossing between A and C may not occur between edges f and e at node u, but has to occur on edge e.

4.1 Preliminaries

Definition 4.1 (Line Ordering Solution)**.** Given an edge $e = \{u, v\} \in E$ labeled with lines $l \in L(e)$, we define $p_e^u(l) = 1, \ldots, |L(e)|$ to give the position of l on e, in clockwise fashion with respect to u. Each $1, \ldots, |L(e)|$ is called a *port* of u for e. If the function $p_e^u : L(e) \mapsto \{1, \ldots, |L(e)|\}$ is bijective (if it is a permutation of $L(e)$), we say p_e^u is a line ordering of e. Correspondingly, $p_e^v(l)$ is the clockwise position of l on e w.r.t. v. We require $\forall l \in L(e) : p_e^u(l) = |L(e)| - p_e^v(l) + 1$. For notational convenience we define the following: $p_e(l) = \{p_e^u(l), p_e^v(l)\}$, $\overleftarrow{p_e^u}(l) = p_e^v(l)$, and $\overleftarrow{p_e^v}(l) = p_e^u(l)$. We denote the set of all line orderings for a given line graph G by σ and call it a line ordering solution.

Figure 4.2, left, gives an example.

4.1.1 Crossing Types

We first lay out the types of crossings that may occur between lines in a metro map. If we assume the graph to be planar, there are exactly 3 types.

1. **Intra-Path Crossings** Let A, B be two lines on edge e. If both A and B extend over a node v to an adjacent edge e' (if they both follow the same local path), a crossing between A and B will occur if $\overleftarrow{p_e^v}(A) < \overleftarrow{p_e^v}(B)$, but $p_{e'}^v(A) < p_{e'}^v(B)$, or vice versa. See Figure 4.3.1 for an example.

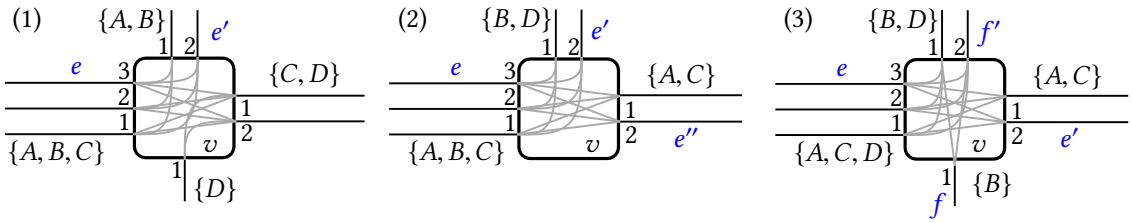


Figure 4.3: Three types of line crossings that may occur in a transit map. (1) If two lines A and B on an edge e extend over a node v into an edge e', a crossing appears if their relative orderings differ. (2) If two such lines extend into two edges e' and e'', whether a crossing appears only depends on the ordering of A and B on e. (3) If A and B locally follow crossing paths, a crossing is unavoidable.

2. **Split Crossings** Let A, B be again two lines on edge e. If A and B extend over a node v to *two* edges e' and e'' (Figure 4.3.2), a crossing occurs if the ordering on e does not match the ordering induced by the circular edge ordering of e' and e''. Let $\pi_e^v(f)$ be the position of f in the circular edge ordering around v, starting in clockwise fashion at e (that is, $\pi_e^v(e) = 1$). If A extends to e'', and B to e', a crossing will occur if $\overleftarrow{p_e^v}(A) > \overleftarrow{p_e^v}(B)$, but $\pi_e^v(e') < \pi_e^v(e'')$, or vice versa. For example, in Figure 4.3.2, a crossing will occur if $\overleftarrow{p_e^v}(A) > \overleftarrow{p_e^v}(B)$.
3. **Ordering-Independent Crossings** Let A be a line extending from an edge e to an edge e', and B a line extending from an edge f to an edge f' (Figure 4.3.3). Additionally, let $\pi_e^v(f) > \pi_e^v(e')$, and $\pi_e^v(f') < \pi_e^v(e')$ (paths (e, e') and (f, f') cross each other at v). A crossing between A and B is induced by the network topology and independent of the actual line ordering.

In the context of this work, we ignore ordering-independent crossings. In particular, this means that we will never count them as crossings.

4.1.2 Metro Line Crossing Minimization

A different formulation of the line crossing minimization problem, called the Metro Line Crossing Minimization (MLCM) was first defined by Benkert et al. in [26]. Instead of line crossings occurring at nodes (as is the case in MLNCM), they occur on edges:

Definition 4.2 (Metro Line Crossing Minimization (MLCM))**.** Given a line graph $G = (V, E, L, \mathcal{L})$. For each set of lines $L(e)$, find (not necessarily compatible) orderings $p_e^v(l)$ and $p_e^u(l)$ for each $e \in E$ such that the global number of line crossings on edges is minimized. For each edge pair adjacent at a node v, their line orderings at v must be compatible (they must not introduce any crossings).

The intuition behind this formulation was that line crossings are obfuscated in the final drawing by stations, and should thus be moved to the edges (Figure 4.2, right). In

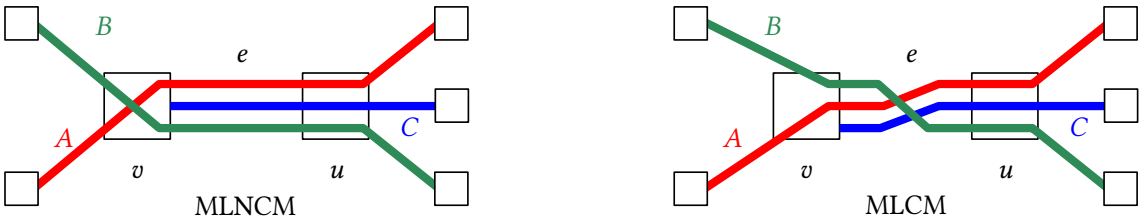


Figure 4.4: A line graph G with line orderings in our MLNCM formulation (left) and the MLCM formulation (right). Under MLNCM, we may move the crossing between lines A and B to node v and "behind" line C ending at v. The optimal ordering has 1 crossing. In contrast, the MLCM problem does not allow a crossing to occur at v, and we have to cross on the edge e. This requires an additional crossing between C with either A or B.

contrast, our MLNCM formulation is conceptually simpler and only allows crossings at nodes. In particular, we do not require the additional constraint of adjacent edges having compatible line orderings. When compared to our formulation, MLCM has several additional drawbacks:

1. In real-world transit maps, lines often cross at non-station nodes where previously paired lines diverge, for example at large intersections in the physical road or rail network. Placing crossings there often looks natural. Such non-station nodes were not considered in the original formulations of MLCM.
2. Real-word transit maps also often have lines crossing at station nodes.
3. Real-world transit maps almost never depict crossings on edge segments.
4. Having two orderings per edge complicates the map rendering process.
5. It is not clear from the ordering where an eventual crossing between two lines should appear on the rendered edge.

At first glance, the two definitions appear similar, with crossings simply moved from nodes (in MLNCM) to edges (in MLCM). Their subtle difference is captured by the following lemma:

Lemma 4.1. *An input line graph G may allow for an optimal line ordering with n crossings under the MLNCM formulation, but only for an optimal number $m > n$ of crossings under the MLCM formulation.*

Proof. Consider the example in Figure 4.4. Under the MLNCM formulation, the optimal number of crossings is 1. The crossing between A and B may "hide" behind line C ending at v, as it may occur in v. Under the MLCM formulation, the optimal number of crossings is 2, as the unavoidable crossing between A and B is not allowed to occur in v, but only on edge e, where a crossing of C with either A or B is unavoidable. □

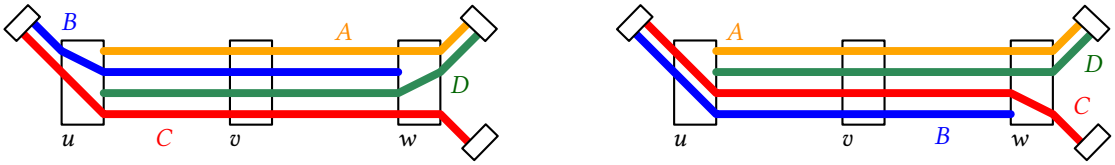


Figure 4.5: Left: A crossing-minimal MLNCM instance. Lines A, B, and D end between pass-through lines at nodes w and u. Right: The periphery condition is fulfilled and lines A, B, and D all end on the outsides of their terminating edge.

4.1.3 Periphery Condition

Previous work often considered the so-called periphery condition for the placement of terminus lines. Under this condition, lines terminating at a node v are required to be positioned at the "outsides" of an edge. This is of always guaranteed for edges with $|L(e)| \leq 2$ and for edges on which *all* lines end at node v.

For example, in Figure 4.5, left, lines B and D end between lines passing through their terminus nodes u and w, which might obfuscate the fact that they end there. In Figure 4.5, right, the periphery condition is enforced and A, B and C end at the outsides of the respective edges. We do not consider the periphery condition in this work, but introduce a different concept (line separations) in Section 4.4, which in many cases (although not all) also ensures the periphery condition.

4.2 Related Work

As mentioned above, previous work in metro-map drawing considered the strongly related Metro Line Crossing Minimization problem (MLCM), where the goal is to find *two* orderings per input edge, one for each side. In this configuration, the problem was introduced by Benkert et al. in [26]. While the authors could not produce a globally optimal ordering, the subproblem of finding the optimal line ordering along a *single* edge was studied. Here, an edge $e = \{u, v\}$ is given together with the lines $L(e)$ that traverse the edge. Additionally, the line orderings at u and v (as induced either by the network topology, or by already found edge orderings for other edges adjacent to u or v) are given. If all $l \in L(e)$ pass through u and v, the orderings at u and v are already fixed, and there is nothing left to do. However, if some lines have u and/or v as a terminus, the problem amounts to finding the optimal insertion order of the lines ending at u and v that in turn induces a crossing-optimal placement of the remaining lines. A $O(n^2)$ dynamic programming algorithm for finding the optimal orderings at u and v was presented, where n is the number of lines passing through u and v. Finding a globally optimal ordering was left as an open problem.

The term MLCM was coined by Bekos et al. in [25] and two subproblems were introduced and studied: In the MLCM problem with line terminals at *station ends* (MLCM under the periphery condition, MLCM-P, sometimes also called MLCM-SE), all lines ending at a terminus node v are placed at the top or the bottom of the ordering, and pass-through lines are placed in-between. In the MLCM problem with terminals at *fixed* station ends (MLCM-PA, sometimes called MLCM-FixedSE), the information whether a line ends at the top or the bottom of an edge is given. By reduction from the Fixed Linear Crossing Number problem, the authors show that already on simple path graphs, MLCM-P is NP-hard. The authors then give a polynomial time algorithm to solve MLCM-PA on simple path graphs and trees. Additionally, they present a polynomial time algorithm to solve MLCM-P on trees if all lines end at nodes of degree 1. General graphs and trees with terminals at internal nodes were not considered.

In [11], Asquith et al. extended the polynomial time algorithm results for MLCM-PA of Benkert et al. to general graphs. They also coined the term *periphery condition* for the requirement that all lines must end either at the top or bottom of edges. For the case where line terminals were not fixed, an ILP was given.

Argyriou et al. introduced another interesting variant of MLCM in [10]. Here it is required that all lines terminate at degree-1 nodes (MLCM-T1). They presented an $O((|E|+|L|^2)|E|)$ algorithm for the general MLCM-T1 problem, and an $O(|V||E| + \sum_{e\in E}(|L(e)|))$ algorithm for the additional special case where each station is represented as a rectangle, and lines can only enter to the left or right of a node. This was further improved by Nöllenburg in [116], who presented an $O(|L|^2 \cdot |V|)$ algorithm for both MLCM-T1 and MLCM-PA which iteratively inserts new lines into an initially empty drawing.

For several years, it was an open problem whether the general unconstrained MLCM problem was also NP-hard. Fink and Pupyrev answered this question in 2013 in the affirmative and provided a reduction proof from MLCM-P [59]. We will extend this proof so that it also works for our MLNCM problem, hence proving its NP-hardness. Fink and Pupyrev also provided a polynomial-time algorithm for deciding whether a general MLCM instance allows for a crossing-free line ordering, and if so, obtain that solution.

Fink et al. observed in [60] that crossings between line bundles may be scattered along one or more edges, without changing the global number of crossings. This may lead to results which are not esthetically pleasing and hard to read. To solve this issue, they grouped crossings into so-called block crossings. A block-crossing is a crossing between neighboring line bundles following the same edges. They showed that the problem of minimizing block crossings was also NP-hard. As an open problem, they stated the problem of distributing block crossings optimally and observed that line bundles following a common sub-path should cross as close to the end of their common sub-path as possible. We will partly replace the concept of block crossings by the more general concept of avoiding *line separations*.

Table 4.1: Overview of the complexities for MLCM and MLNCM subproblems.

Problem	Graph	Compl.	Proof	Description
MLCM	general	NP-hard	[59]	*No restrictions, minimize number of line crossings.*
	path	linear	(trivial)	
MLCM-P	general	NP-hard	[25]	*Terminus lines must be grouped at the top or bottom of edges.*
	path	NP-hard	[25]	
	tree[2]	polynomial[1]	[25]	
MLCM-PA	general	linear[1,2]	[116]	*Like MLCM-P, but line terminus positions (either top or bottom) are fixed.*
MLCM-T1	general	linear[1,2]	[116]	*All lines must terminate at degree 1 nodes.*
MLNCM	general	NP-hard	Thm. 4.3	*No restrictions, minimize number of line crossings.*
	path	linear	(trivial)	
	tree-like[1,2]	linear[3,4]	Cor. 4.51	
	2-tree-like[1,2]	linear[3,4]	Cor. 4.56	
MLNCM-W	general	NP-hard	Thm. 4.4	*Minimize weighted line crossings.*
	general[1,2]	NP-hard	Thm. 4.9	
	path	linear	(trivial)	
MLNCM-S	general	?	—	*Minimize line crossings and separations.*
	tree-like[1,2]	linear[3,4]	Cor. 4.49	
	2-tree-like[1,2]	linear[3,4]	Cor. 4.54	
MLNCM-WS	general	NP-hard	Thm. 4.5	*Minimize weighted line crossings and weighted separations.*
	general[1,2]	NP-hard	Cor. 4.10	

[1] Lines follow simple paths. [2] Lines always terminate at degree-1 nodes. [3] The maximum number of lines M is considered a constant. [4] The maximum node degree D is considered a constant.

Our own MLNCM formulation was introduced in [19], along with two ILPs to solve it. As mentioned above, our formulation imposes a single line ordering on each edge and only allows crossing to happen on nodes. Two additional variants were presented: a weighted version of MLNCM, where line crossings may be weighted per node, and a version which additionally penalized line separations (see the previous paragraph). Several simplification rules were given which may be applied to the line graph prior to finding the optimal line ordering, while maintaining optimality. This was further extended by several so-called untangling rules in [20]. For an overview over the subproblems of MLCM and MLNCM, as well as their complexities, see Table 4.1.

In a recent work, Frank et al. [62] studied the drawing of hypergraphs as metro-maps. In this context, they introduced a variant of our MLNCM problem, namely the Metro Line Crossing Node Minimization Problem (MLCNM). Here, the goal is not to minimize the global *number of crossings*, but the *number of crossing nodes* (nodes where one or more crossings appear). Then, of course, an optimal solution could place all crossings at a single node, making the resulting map very hard to read. Even in this configuration,

the problem remains NP-hard; the authors provide an NP-hardness proof via reduction from Vertex Cover.

As mentioned above, problem formulations that are similar to our MLNCM formulation previously appeared in the context of wire routing in electronic designs [75] and edge bundling [123], although they considered only the simple case were all lines end at nodes of degree 1. For this setting, Pupyrev et al. described an algorithm similar to our greedy-lookahead algorithm in Section 4.6.4 [123].

The problem is also related to drawing a (multi-)graph with a minimum number of edge crossings. The major differences are that (1) in these works, the goal is to find a graph embedding with a minimum number of crossings, and (2) crossings of *paths* are not typically not considered. In the case of MLCM and MLNCM, the embedding is fixed and crossings between entire edges are not considered (if edges cross in the input embedding, the embedding can be made planar by adding additional crossing nodes).

4.3 Weighted MLNCM

It is sometimes desirable to move crossings to preferred locations. In our definition of a line graph, we explicitly allowed non-station nodes, and we would like to prefer crossings to happen at large non-station nodes, to ensure easy readability. We might also want to make crossings for a single line more expensive, for example because the line should be rendered with greater width. We hence also consider the following variation of MLNCM:

Definition 4.3 (Weighted Metro Line Node Crossing Minimization (MLNCM-W))**.** Given a line graph $G = (V, E, L, \mathcal{L})$ and a weight function $w : \mathcal{L}^2 \times V \mapsto \mathbb{R}_0^+$, with $w(l_1, l_2, v) = w(l_2, l_1, v)\ \forall l_1, l_2 \in \mathcal{L}^2\ \forall v \in V$. A crossing of two lines l_1, l_2 at a node v is weighted by $w(l_1, l_2, v)$. The goal is then to find a permutation for each set of lines $L(e), e \in E$ such that the sum of weighted line crossings between adjacent edges is minimized.

Obviously, MLNCM is a sub-problem of MLNCM-W, with $w(l_1, l_2, v) = 1$ for each $v \in V$ and each $l_1, l_2 \in \mathcal{L}^2$.

4.4 Minimizing Line Separations

Minimizing line-crossings alone might not always produce esthetically pleasing line orderings. We consider Figure 4.6. For each example, the number of line crossings is 4, which is indeed optimal. The esthetic quality, however, greatly differs. In particular, the top two examples make it very hard to follow line courses. Additionally, the unavoidable crossing between line bundle $\{A, B\}$ and line bundle $\{C, D\}$ is depicted more clearly

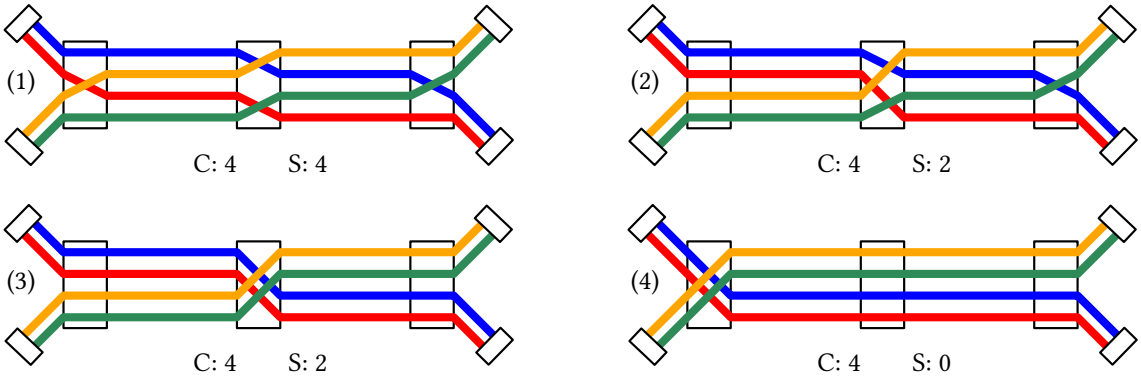


Figure 4.6: All 4 line ordering examples have 4 crossings (C), which is optimal. However, (1) and (2) are very hard to read not visually appealing. While (3) looks better, the crossing of the two line bundles is best depicted in (4). Counting line separations (S) exactly captures this intuitive preference.

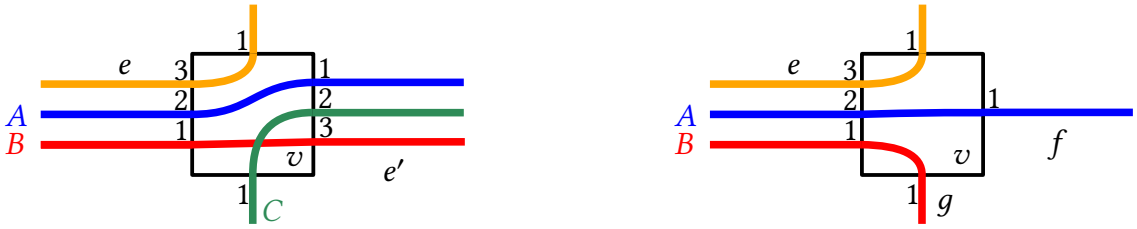


Figure 4.7: Left: lines A and B are separated in v by C. Right: the branching of A and B into two different edges f and g does not count as a separation.

in the lower right version. In this section, we formalize the intuitive preference for the lower right version using a single additional new concept not considered so far in the literature: *line separations*. We also describe several additional problems that can be solved via penalizing line separations. Informally, a line separation occurs at a node v between edges e and f if lines l_1 and l_2 were placed next to each other on edge $e = \{u, v\}$, but not anymore on edge $f = \{v, w\}$. Figure 4.7 gives an example. Formally:

Definition 4.4 (Line Separation)**.** Let $e = \{u, v\}$ and $f = \{v, w\}$ be two edges adjacent at a node v, and let l_1 and l_2 be two distinct lines on e and f, that is $l_1, l_2 \in L(e)$, $l_1, l_2 \in L(f)$, and $l_1 \neq l_2$. If $|p_e^v(l_1) - p_e^v(l_2)| = 1$ and $|p_f^v(l_1) - p_f^v(l_2)| \neq 1$ or vice versa, we say lines l_1 and l_2 separate in v.

In Figure 4.6, (1) has 4 line separations, (2) has 4 line separations, (3) has 2 line separations, and (4) has zero line separations. A ranking based on both the number of line crossings and line separations therefore exactly matches our intuitive preference, and (4) is the clear winner.

We now reconsider the periphery condition example from Figure 4.5. The solution

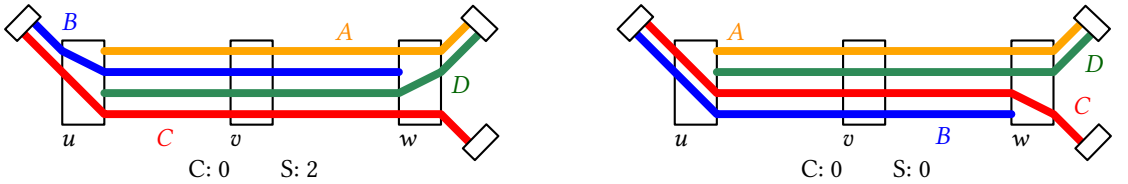


Figure 4.8: Left: Crossing-free line ordering, the periphery condition is violated two times. Right: If line separations are also minimized, the ordering satisfies the periphery condition.

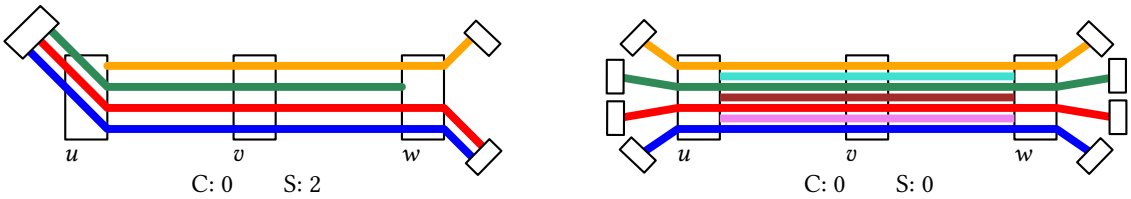


Figure 4.9: Left: Separation-free line ordering, but periphery condition is violated. Right: Extreme example of a crossing and separation free ordering, in which the periphery condition is violated 6 times.

violating the periphery condition has two separations, the solution satisfying the periphery condition has 0 separations (Figure 4.8). However, avoiding line separations might not always lead to solutions which satisfy the periphery condition. As a counterexample, consider Figure 4.9, left. The separation-free solution still violates the periphery condition, and there would be an alternative separation and crossing free line ordering which ensure the periphery condition by placing the green line at the bottom. An extreme case can be seen in Figure 4.9, right.

We now define the following extension of our original MLNCM problem:

Definition 4.5 (Metro Line Node Crossing and Separation Minimization (MLNCM-S))**.** Given a line graph $G = (V, E, L, \mathcal{L})$. For each set of lines $L(e)$, find a single ordering for each $e \in E$ such that the global number of line crossings and line separations between adjacent edges is minimized.

Just like for MLNCM, we would like to be able to prefer certain nodes for line separations to happen, or to give line separations a higher weight. We hence also consider the following problem, of which all MLNCM variants presented so far are a subproblem:

Definition 4.6 (Weighted Metro Line Node Crossing and Separation Minimization (MLNCM-WS))**.** Given a line graph $G = (V, E, L)$, a crossing weight function $w_\times : \mathcal{L}^2 \times V \mapsto \mathbb{R}_0^+$, and a separation weight function $w_{||} : \mathcal{L}^2 \times V \mapsto \mathbb{R}_0^+$. Again we require $w_\times(l_1, l_2, v) = w_\times(l_2, l_1, v)$ and $w_{||}(l_1, l_2, v) = w_{||}(l_2, l_1, v)\ \forall l_1, l_2 \in \mathcal{L}^2\ \forall v \in V$. For each set of lines $L(e)$, find a permutation for each $e \in E$ such that the global sum of weighted line crossings and line separations between adjacent edges is minimized.

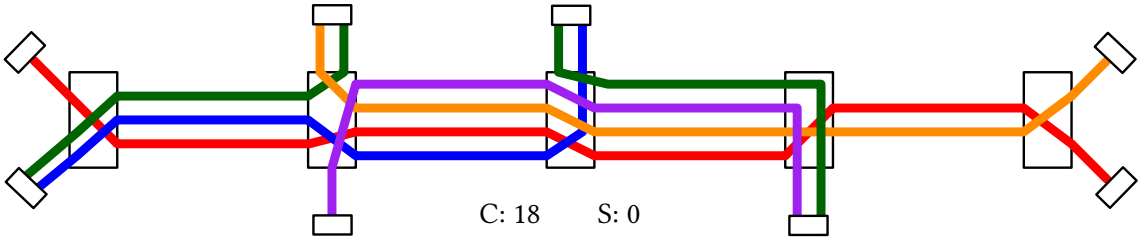


Figure 4.10: Optimal line ordering when only line separations are considered. The resulting ordering is not very readable, many crossings could even be avoided without adding new line separations.

At this point, a valid question might be whether we need to count crossings at all if we consider line separations - would it maybe be sufficient to only penalize line separations? Figure 4.10 gives a clear counter-example. Although the depicted line ordering is completely separation free, it is obviously not optimal, and many crossings could even be avoided without introducing additional line separations.

4.5 NP-Hardness Results

A direct consequence of Lemma 4.1 is that there is no straightforward way to translate an optimal MLNCM solution to an optimal MLCM solution, impeding a simple NP-hardness proof for MLNCM by reduction from MLCM. Fortunately, the original NP-hardness proof of MLCM can be extended to also work for MLNCM and MLNCM-W, which we lay out in the following.

We now say a line l uniquely extends from an edge e over a node v if l does not branch at v, and first prove the following lemma:

Lemma 4.2. *Given a line graph $G = (V, E, L, \mathcal{L})$, a line ordering solution σ, and an edge $e = \{u, v\}$, where a bundle $\mathcal{B}$ of lines in $L(e)$ (lines that are positioned next to each other) uniquely extends over u and v into edges $f = \{t, u\}$ and $g = \{v, w\}$. Figure 4.11 gives an example. We can group any crossings that occur in u or v between lines in $\mathcal{B}$ in one of u or v without creating new crossings.*

Proof. We first observe that by swapping lines in $\mathcal{B}$ on e, we might only introduce new crossings at u or v between lines in $\mathcal{B}$ itself. Hence we can ignore other lines. All unavoidable line crossings that must occur in u and v are induced by the ordering of $\mathcal{B}$ in f and g. If we set the ordering of $\mathcal{B}$ on e so that it is compatible with the ordering of f, only the unavoidable crossings will occur in v, and no crossings occur in u. If we set the ordering of $\mathcal{B}$ on e so that it is compatible with the ordering of g, only the unavoidable crossings will occur in u, and no crossings occur in v. As the summed

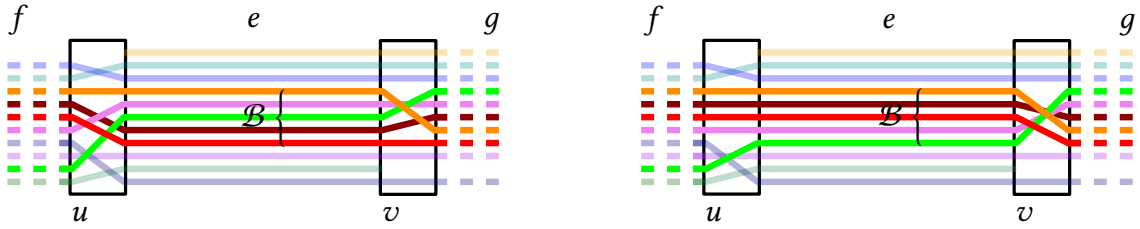


Figure 4.11: Line bundle $\mathcal{B}$ extends over both u and v. All crossings that occur in u or v between lines in $\mathcal{B}$ (left) can be clustered in u or v (right) without creating any new crossings.

number of crossings in u and v is now exactly the number of unavoidable crossings induced by the ordering on f and g, no new crossings could have been added. □

Theorem 4.3. *The decision variant of MLNCM is NP-hard.*

Proof. Similar to the original NP-hardness proof of MLCM, our proof is a reduction from MLCM-P (the Metro Line Crossing Minimization problem under the periphery constraint), shown to be NP-hard even on path graphs in [25]. We consider an input line graph $G = (V, E, L, \mathcal{L})$, which is a path. Figure 4.12, left shows a single edge e of such a path graph with a crossing-optimal ordering for MLCM under the periphery condition (the ordering of lines extended over u and v depends on the rest of the graph). To translate this problem into an MLNCM instance, we now build an extended graph $G' = (V', E', L', \mathcal{L}')$, in which we first split each edge e into a left edge $e_1 = \{u, x_e\}$, and a right edge $e_2 = \{x_e, v\}$, where x_e is a new node which models crossing on the original e in our MLNCM formulation. To ensure that lines ending at either u or v are always placed at the top or bottom, we now employ a technique from [25]: we add nodes u_t^e and u_b^e at the top and bottom of u, and nodes v_t^e and v_b^e at the top and bottom of v (Figure 4.12, right). We then add a bundle $\mathcal{A}$ of unique lines following path (u_t^e, x_e, v_b^e) and a similar bundle $\mathcal{B}$ of again unique lines following path (u_b^e, x_e, v_t^e), with $|\mathcal{A}| = |\mathcal{B}| = \ell$. Obviously, $\mathcal{A}$ and $\mathcal{B}$ have to cross either at u, x_e or v, adding ℓ^2 unavoidable crossings per initial input edge e to any crossing-minimal solution. Additionally, each line extending over both u and v also has to cross $\mathcal{A}$ and $\mathcal{B}$, again offsetting any crossing-minimal solution by $n_e \cdot 2\ell$, where n_e is the number of lines on e that do not end at either u or v. However, lines ending at u *or* v may circumvent a crossing with either $\mathcal{A}$ or $\mathcal{B}$ by ending at the top or bottom of the respective node (see again (Figure 4.12, right). If a line l ends at *both* v and u, it does not have to cross any line in the original graph G, and we can thus ignore such lines without affecting optimality. If we now choose ℓ large enough such that crossing $\mathcal{A}$ or $\mathcal{B}$ always leads to a non-optimal solution, any solution violating the periphery constraint is not optimal. However, our MLNCM formulation would allow $\mathcal{A}$ and $\mathcal{B}$ to cross, for example, at u instead of x_e. Then a terminating line could violate the

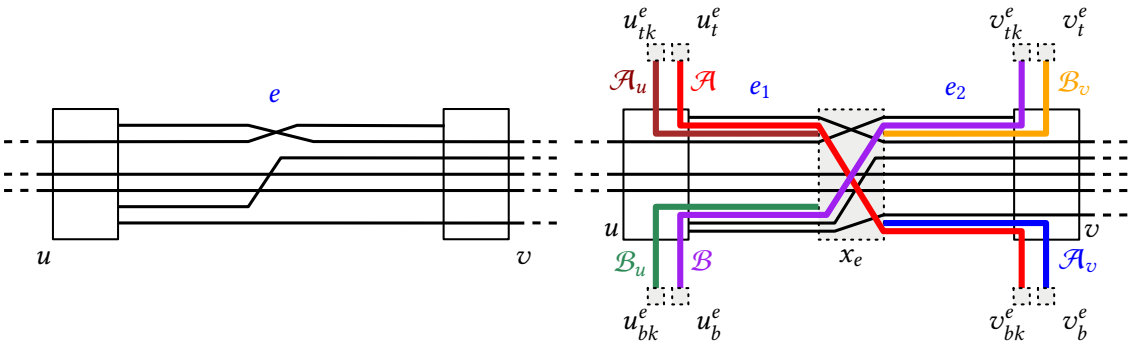


Figure 4.12: Left: Part of an MLCM-P instance on a path graph G. Lines ending at u and v are required to end either at the top or bottom of the node. The ordering of pass-through lines depends on the rest of the graph. Right: An equivalent MLNCM instance on an extended graph G'. We model crossings on edges by an additional node x_e which splits edges e into a left edge e_1 and a right edge e_2. To ensure the periphery condition, we add four additional nodes u_t, u_b, v_t and v_b and add sufficiently large line bundles $\mathcal{A}$ and $\mathcal{B}$ in such a way that each pass-through line has to cross them all, but lines ending at a station may circumvent the crossings if they end at the top or bottom. The additional keeper nodes u_{tk}, u_{bk}, v_{tk}, v_{bk} and keeper line bundles $\mathcal{A}_u$, $\mathcal{B}_u$, $\mathcal{A}_v$, and $\mathcal{B}_v$ are needed to prevent a crossing between the bundles at nodes u and v.

periphery condition, while still only crossing one of $\mathcal{A}$ and $\mathcal{B}$. To prevent this, we further extend G' and add two *keeper* nodes u^e_{tk} and u^e_{tb} to u, and two keeper nodes v^e_{tk} and v^e_{tb} to v. For u, they are placed to the left of u^e_t and u^e_b, for v, they are placed to the right of v^e_t and v^e_b. We then add *keeper bundles* $\mathcal{A}_u$ following path (u^e_{tk}, u, x_e), $\mathcal{A}_v$ following path (v^e_{bk}, v, x_e), $\mathcal{B}_u$ following path (u^e_{bk}, u, x_e) and $\mathcal{B}_v$ following path (v^e_{tk}, v, x_e), all with ℓ lines. Any crossing of lines extending over u and v with any keeper bundle is unnecessary. Additionally, any crossing of keeper bundles with $\mathcal{A}$ and $\mathcal{B}$ is also unnecessary - it can be avoided by $\mathcal{A}$ and $\mathcal{B}$ crossing at x_e. If ℓ is large enough, the keeper bundles therefore enforce that $\mathcal{A}$ and $\mathcal{B}$ cross at x_e, and thus ensure the periphery condition. At each node, each line pair may cross at most once, so this is obviously satisfied for $\ell = |V| \cdot |\mathcal{L}|^2$. We therefore set $\ell = |\mathcal{L}|^2$. Any avoidable crossing with $\mathcal{A}$, $\mathcal{B}$ or the keeper bundles then adds more crossings to G' than could be saved between original lines in $\mathcal{L}$. Any optimal MLNCM solution of G' hence satisfies the periphery condition. We note that the lines extending over u and v may now still cross at u or v, which is not allowed under the MLCM formulation. However, because of the enforced periphery condition, we may indeed only encounter crossings between such pass-through lines. If such pass-through lines are present in a node v, they also pass through an adjacent crossing node x_e, and therefore Lemma 4.2 applies to v and x_e. We can thus always move any crossings on v to x_e without affecting optimality. Any optimal MLNCM solution for G' can thus be transformed into a feasible and optimal MLCM-P solution for G.

More specifically, any crossing-optimal solution for G' is offsetted by $K = |E| \cdot \ell^2 +$

$\sum_{e \in E}(n_e + |L(e)| \cdot \ell)$. If there is an optimal MLNCM solution for G' with $k + K$ crossings, there is hence an optimal MLCM-P solution for G with k crossings, and vice versa. □

Theorem 4.4. *The decision variant of MLNCM-W is NP-hard.*

Proof. Obviously, we can reduce an MLNCM instance to MLNCM-W if we set all weights to 1. □

Theorem 4.5. *The decision variant of MLNCM-WS is NP-hard.*

Proof. Again, we can reduce any MLNCM instance to MLNCM-WS if we set all weights for crossings to 1, and for separations to 0. □

Theorem 4.6. *The decision variants of MLNCM, MLNCM-W, and MLNCM-WS are NP-complete.*

Proof. We can obviously count the crossing and separation score of a given solution in polynomial time by comparing the positions of all line pairs on all adjacent edges. As MLNCM, MLNCM-W and MLNCM-WS are NP-hard, they are thus also NP-complete. □

Note that an NP-hardness proof for MLNCM-S, that is the special case of MLNCM-WS where both crossings and separations are always weighted by 1, is missing from this section. Unfortunately, the NP-hardness proof of MLNCM from above does not work for MLNCM-S anymore. The dilemma is that if we solve the graph G' with MLNCM-S, it does not seem to be guaranteed that the resulting edge ordering is crossing-minimal (that is, it might be that we introduced an additional crossing to save a separation). We consider the complexity of MLNCM-S an interesting open problem.

4.6 Heuristic Approaches

Motivated by the NP-hardness results from above, and as alternatives to the ILP formulation given below, this section describes several baseline heuristic approaches to find good solutions for MLNCM-WS (and therefore its subproblems MLNCM, MLNCM-W, and MLNCM-S). All heuristic approaches will be evaluated in Section 4.11.2. As most of the heuristic approaches have to repeatedly calculate the target function, we first quickly describe how it can be efficiently determined for MLNCM, MLNCM-S, and two relevant special cases of MLNCM-W and MLNCM-WS.

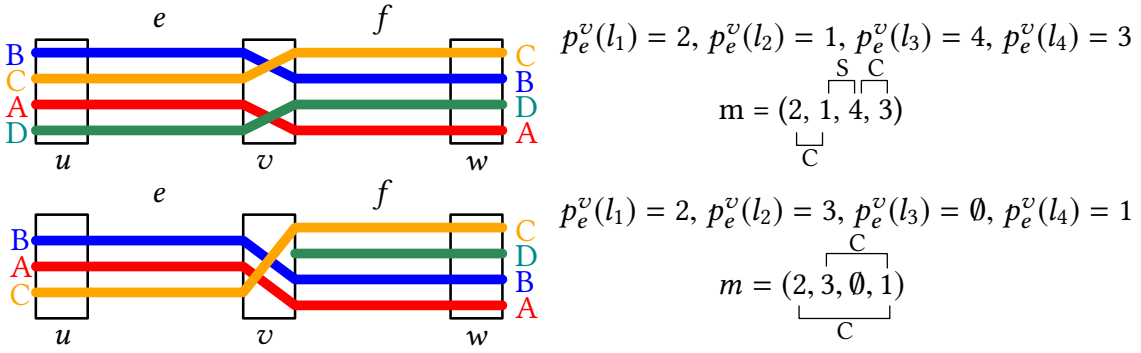


Figure 4.13: Counting the number of crossings between the lines on two edges e and f is essentially equivalent to counting the number of inversions of a list $m = (p_e^v(l_1), ..., p_e^v(l_{|L(f)|}))$ where l_i is the line on f at position i w.r.t. w, and $p_e^v(l)$ is the position of line l on e w.r.t v. The number of separations introduced at e (not at f) is the number of pairs i and $i+1$, with $|m[i] - m[i+1]| > 1$. Top: The straightforward variant where $L(e) = L(f)$. Bottom: If $L(e) \neq L(f)$, we have to insert placeholders into m to not distort the number of separations (note that no separation between lines A and B is induced at v in e, as lines A and B were already separated in f by D).

4.6.1 Efficient Calculation of the Target Function

We first consider the case of calculating the score of an MLNCM-S instance at a single degree-2 node v adjacent to edges e and fs with $L(e) = L(f)$ (Figure 4.13). The naive way to count the number of crossings and separations at v would be to take all $O(|L(e)|^2)$ pairs of different lines and compare their position in e to their position in f.

A more sophisticated approach is to take e as a reference, and consider the ordering of f with respect to the ordering of e. Let $p_e^v(l)$ be the position of line l on e w.r.t. v, and let l_i be the line at position i on f w.r.t. w. Then the number of crossings at v is simply the number of inversions (that is, the number of pairs which are not in order) in the list $m = (p_e^v(l_1), p_e^v(l_2), \ldots, p_e^v(l_{|L(f)|}))$.

Given some list a of n elements on which a total order exists, the number of inversions $\text{inv}(a)$ in a can be calculated in $O(n \log n)$ with a small extension to the merge sort algorithm [96]: if we merge two (ordered) lists a_l (the left list) and a_r (the right list) using the standard linear time algorithm, each time we encounter an element in a_l at position i which is *greater* than the current element in a_r at position j, we know that all elements to the right of $a_l[i]$ are also greater than $a_r[j]$, and therefore we have $|a_r| - i$ inversions. Summing these local inversion numbers over all merge steps gives the total number of inversions in quasilinear time.

The number of *separations* caused at v in e is simply the number of adjacent elements in m for which $|m[i] - m[i+1]| > 1$ and can be determined in linear time. For the number of separations caused at v in f, we use the same approach, but now with f as a reference.

This approach can be extended to also work on adjacent edges e and f if $L(e) \neq L(f)$: if no $p_e^v(l)$ exists because line l is only present in f, but not in e, we simply ignore it. However, this approach would now introduce erroneous separations as two lines which were not next to each other in e might now appear next to each other in m at positions i and $i+1$, with $|m[i] - m[i+1]| > 1$. To solve this, we build a second list m' in which we insert a placeholder value for each line $l \notin L(e)$ and only consider adjacent line pairs in which none of the elements is a placeholder (see Figure 4.13 for an example).

It is easy to see that this approach can be used to count all intra-path crossings and separations at v by simply applying the method above to all $O(\deg(v)^2)$ pairs of different edges. To also count *split crossings* in which two lines branch at v for a reference edge e, we build a list m^* similar to the list m described above, but now based on the adjacent edges in clockwise order, and again count the number of inversions. In particular, note that we also count intra-path crossings this way.

To finally get distinct values for the number of split crossings, intra-path crossings, and separations, we simply subtract the number of intra-path crossings determined from the individual m from $\text{inv}(m^*)$.

For a single node v, we can therefore calculate distinct values for the number of split crossings, intra-path crossings, and separations, in $O(D^2 M \log M)$, where D is the maximum node degree, and M is the maximum number of lines, an improvement from the naive $O(D^2 M^2)$ method described above.

This approach does not work anymore if arbitrary crossing and separation weights are present for line pairs. If, however, crossing and separation weights are uniform *per node and crossing type*, we can calculate the target function like described above. An additional observation will prove to be useful for the local search approaches described below: if we adjust the line ordering at some node $e = \{u, v\}$, this only affects the number of orderings or separations introduced at u and v. To compare the target function value before and after a local change at an edge e, it is therefore enough to evaluate the target function only at v and u.

4.6.2 Exhaustive Search

Surely the most naive approach to find an optimal line ordering is to do an exhaustive search. For each edge e, there are $|L(e)|!$ line permutations. An exhaustive search enumerates all $\prod_{e \in E} |L(e)|!$ possible line ordering solutions and keeps track of the best solution found so far (possibly with an early stop if the minimum score of 0 is encountered).

4.6.3 Greedy Search

Given a line graph G, define an initial ordering on the edges E. Iterate over E in this order. For an edge $e = \{u, v\} \in E$, choose a *reference node* (either u or v). W.l.o.g. let u by the reference node. At u, a strict weak ordering $<$ on the lines $L(e)$ is induced by the adjacent edges not equal to e. For two lines $l_1, l_2 \in L(e)$, there are three possibilities: (1) l_1 and l_2 branch at u into two edges f and g, then $<$ depends on the circular ordering of f and g at u. (2) l_1 and l_2 follow the same edge f (and no other), then $<$ depends on the ordering of $L(f)$ (if settled), or they are incomparable otherwise. (3) l_1 and l_2 do not extend together over u, or one or both of them branch into multiple edges, then we mark them as incomparable. If two lines $l_1, l_2 \in L(e)$ are incomparable at u, but comparable at v, base $l_1 < l_2$ on v. Otherwise, base $l_1 < l_2$ on some fixed global ordering (for example, based on the lexicographic ordering of the line names). Sort $L(e)$ according to $<$ and continue with the next edge.

4.6.4 Greedy Search with Lookahead

An obvious extension of the Greedy Search approach is the following: If in case (2) the ordering of $L(f), f = \{u, w\}$ is not settled, check whether an ordering of l_1 and l_2 is induced at w, and so on, following the path of l_1 and l_2 until they branch, or until an edge with an already fixed ordering is found. Algorithms following a similar idea have been described in [75] and [123].

Lemma 4.7. *Given a MLNCM instance of a line graph G in which each $l \in \mathcal{L}$ follows a simple path, and where each line terminates at a node of degree 1. The Greedy Branch-Based algorithm will find a crossing-minimal line ordering solution, regardless of the initial ordering of E.*

Proof. Assume no line ordering has been settled yet, and we begin with an edge $e = \{u, v\}$. Let u be the reference node. If we follow a line pair $l_1, l_2 \in L(e)$ over u, then there are exactly two possibilities: either l_1 and l_2 will branch at some point, inducing an ordering, or they will terminate together at a node t. If they terminate together at t, they are incomparable, and we can check whether the other node v induces an ordering of them. If not, they also terminate on the v-side, and thus follow the exact same path through G. Their relative ordering is therefore irrelevant, and we can treat them as a single line.

This induced total ordering of $L(e)$ w.r.t. u is optimal: on the u-side, it allows for orderings without any crossings between lines in $L(e)$. On the v-side, any induced crossing is caused by two lines l_1 and l_2 branching into two edges. But as the ordering of l_1 and l_2 was already induced by a branch on the u-side, any such crossing is unavoidable. For each subsequently processed edge f, if we follow a line pair $l_1, l_2 \in L(f)$, a third possi-

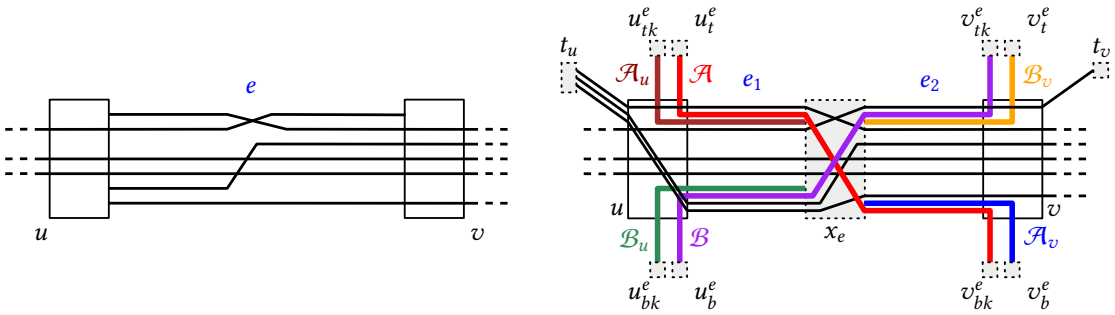


Figure 4.14: Illustration of the proof for Theorem 4.9.

bility appears: we arrive at an edge e with an already settled (optimal) line ordering, and the same argument holds. □

In the worst case, this algorithm has the check for each edge $|\mathcal{L}|^2$ line pairs. As we follow each line pair for each edge through the graph on the search for a branch point or a settled edge, it looks like our algorithm is also quadratic in the number of edges. Note, however, that after a search for a branch point or a settled edge for two lines l_1 and l_2, starting at an edge e and ending at an edge f, we can already store the relative ordering of l_1 and l_2 for each edge in between. For each line pair, we therefore only have to process an edge *once*, resulting in a worst-case complexity of our greedy branch-based algorithm of $O(|E| \cdot |\mathcal{L}|^2 \cdot D)$ (D is the maximum node degree and is needed as we have to check D adjacent edges for a branch in the worst case).

Corollary 4.8. *Given a MLNCM instance of a line graph G in which each $l \in \mathcal{L}$ follows a simple path, and where each line terminates at a node of degree 1. The Greedy Branch-Based algorithm will find a crossing-minimal line ordering solution, regardless of the initial ordering of E, in linear time, if $|\mathcal{L}|$ and D are considered constants.*

This matches previous results in the MLCM setting (MLCM-T1, see Table 4.1).

Weighted Case and Line Separations

To also consider crossing weights in our greedy search with lookahead, we always chose the ordering of an edge e w.r.t. the adjacent node for which the summed crossing costs are more expensive. For the special case from Corollary 4.8, the greedy search with lookahead will no longer produce an optimal solution if line crossings are weighted. In fact, the following holds:

Theorem 4.9. *The decision variant of MLNCM-W remains NP-hard on graphs where all lines follow simple paths and terminate in nodes of degree 1.*

Proof. We slightly extend the proof for Theorem 4.3 and add a termination node t_u for each terminus node u with $\deg(u) > 1$, connected to u by an edge holding all lines T terminating at u. We then set the crossing weights of each line $l \in T$ with any other line at u to 0. Figure 4.14 provides an example. □

Corollary 4.10. *The decision variant of MLNCM-WS remains NP-hard on graphs where all lines follow simple paths and terminate in nodes of degree 1.*

As we only consider the relative ordering of line pairs down the path on both sides, we also cannot check in general whether an ordering will introduce any line separations later on (it might depend on the relative ordering of lines not on e which will later join other edges down the path). Line separations are thus not considered in our greedy search with lookahead.

4.6.5 Steepest-Ascent Hill Climbing

Let $\sigma \in \Omega$ be a line ordering solution, and let $N(\sigma)$ be the *neighborhood* of σ. We recall from Section 1.3.4 that steepest-ascent hill climbing iteratively chooses the neighbor σ' with the best (in our case: minimal) score $\Theta(\sigma')$.

An intuitive neighborhood definition is the following: for a solution σ, $N(\sigma)$ is the set of solutions obtainable by changing the ordering of only a *single* edge. This variant has been evaluated in [19]. However, even then the neighborhood size depends on the sum of factorials $|L(e)|!$, and we have to enumerate all of them to find the best one. In [19], we fell back to a stochastic approach and only checked 1,000 random line orderings of an $L(e)$ if $|L(e)|$ was too large (if $|L(e)|! > 3 \cdot 10^5$). In this work, we consider an even simpler neighborhood: $N(\sigma)$ is the set of solutions obtainable by swapping a single line pair on a single edge. Then the neighborhood size is $O(|E|M^2)$, where again $M = \max_{e \in E} |L(e)|$.

4.6.6 Simulated Annealing

We again have a neighborhood $N(\sigma)$, the set of solutions obtainable from σ by changing the ordering of a single line pair on a single edge. We always keep a neighbor σ' if its score is better than the current one, that is $\theta(\sigma') < \theta(\sigma)$. To get out of local optima, we additionally add a temperature T which decreases with each iteration. If the randomly selected neighbor σ' is *worse or equal* to the current one at iteration i (that is $\theta(\sigma') \geq \theta(\sigma)$), we still select it with probability $P(\sigma, \sigma', T_i)$. Following Kirkpatrick et al. [94], we set $P(\sigma, \sigma', T_i) = \exp((-\theta(\sigma') - \theta(\sigma))/T_i)$, where T_i is the current annealing temperature at iteration i. Unfortunately, there are no generally accepted best choices of the temperature scheduling and/or the initial temperature T_0. In [19], we chose $T_0 = 1,000$ and $T_i = T_0/i$.

4.7 Baseline ILP Formulation

To achieve optimal solutions and to also measure the approximation errors of the heuristics described above, we would like to have an ILP that exactly optimizes a given MLNCM, MLNCM-W, MLNCM-S, or MLNCM-WS instance. This section first provides an introductory baseline ILP formulation for MLNCM, MLNCM-W, and MLNCM-WS in which each possible line crossing (or separation) is enumerated. We will improve this ILP in Section 4.8 by removing the need for this explicit enumeration.

4.7.1 Line Positions

To later retrieve the position $p_e(l)$ for a line $l \in L(e)$ on an edge e, we add $|L(e)|^2$ line position variables $x_{elp} \in \{0,1\}$ for each edge $e \in E$, where $p = 1, \ldots, |L(e)|$ is a position slot on e. We would like to have $x_{elp} = 1$ when line l is assigned to slot p on e, or $x_{elp} = 0$ if not. For each edge $e = \{u, v\}$, we store whether the variables x_{elp} are given w.r.t. u, or w.r.t. v. To enforce that each line l is assigned to exactly one position p, we then add the following set of constraints:

$$\forall l \in L(e) : \sum_{p=1}^{|L(e)|} x_{elp} = 1. \tag{4.1}$$

Equation 4.1 is still fulfilled if all lines are assigned the same position slot. To avoid such slot overfills, we additionally guarantee that each position slot is assigned exactly one line with the following set of constraints:

$$\forall p \in \{1, ..., |L(e)|\} : \sum_{l \in L(e)} x_{elp} = 1. \tag{4.2}$$

4.7.2 Counting Line Crossings

To count crossings in the objective function, we have to consider the two relevant crossing types described in Section 4.1.1: intra-path crossings, and split crossings.

Intra-Path Crossings

In the case of intra-path crossings, two lines A and B on an edge $e = \{u, v\}$ both extend over w into an edge $f = \{v, w\}$. Then A and B cross at v if $p_e^v(A) < p_e^v(B)$ and $p_f^v(A) < p_f^v(B)$, or $p_e^v(A) > p_e^v(B)$ and $p_f^v(A) > p_f^v(B)$. We would now like to have a binary decision variable $x_{efAB} \in \{0,1\}$ which is 1 if a crossing is induced by A and B at v, or 0 otherwise. We then add x_{efAB} to the objective function, possibly weighted if we have

an MLNCM-W instance. To enforce this assignment, we simply enumerate all possible crossing positions for A and B. Note that we have to consider whether the position variables on e and f are given w.r.t. v, or w.r.t. to u or w. If they are given w.r.t. to v for both e and f, a crossing would for example occur if $x_{eA1} = 1, x_{eB2} = 1, x_{fA1} = 1$, and $x_{fB2} = 1$, as this would imply $p_e^v(A) < p_e^v(B)$ and $p_f^v(A) < p_f^v(B)$. If the positions on either e or f are not given w.r.t. to v, a crossing would for example occur if $x_{eA2} = 1, x_{eB1} = 1, x_{fA1} = 1$, and $x_{fB2} = 1$. For each such crossing-inducing variable assignment, we then add constraints like the following for the former example:

$$x_{eAi} + x_{eBj} + x_{fAi'} + x_{fBj'} - x_{efAB} \leq 3, \tag{4.3}$$

where i, j and i', j' are two position pairs which induce a crossing. If all variables x_{eA1}, x_{eB2}, x_{fA1}, and x_{fB2} are set to 1, the only way to fulfill the ≤ 3 constraint is to set x_{efAB} to 1. If only one of the variables is 0, x_{efAB} will be 0 if it appears in the objective function with a positive coefficient. If the coefficient is 0 (because crossings have been weighted with 0), its value is irrelevant.

Split Crossings

If A extends over v into an edge f, and B extends into an edge g, their relative positions on one side of v are fixed by the circular edge ordering of f and g, and their actual position on f or g is irrelevant. Whether a crossing occurs only depends on their ordering in e.

For such cases, we introduce a binary decision variable $x_{efgAB} \in \{0, 1\}$ which should be 1 if the positions of A and B on e induce a crossing at v. Similar to Equation 4.3, we then enumerate all position pairs (i, j) of A and B on e which induce a crossing and add a constraint of the following form for each of them:

$$x_{eAi} + x_{eBj} - x_{efgAB} \leq 1. \tag{4.4}$$

Like before, if the first two variables are 1 and the crossing inducing assignment was chosen, the split crossing decision variable x_{efgAB} has to be 0 to fulfill the constraint.

4.7.3 Counting Line Separations

To consider line separations, we introduce a binary decision variable $x_{efA||B}$ which should be 1 if lines A and B separate when extending from e to f. Again, we enumerate all position pairs (i, j, i', j') of A and B on e and f which induce such a separation. We then enforce the desired assignment by adding the following constraint per position pair:

$$x_{eAi} + x_{eBj} + x_{fAi'} + x_{fBj'} - x_{efA||B} \leq 3. \tag{4.5}$$

4.7.4 ILP Size

For each edge, we need exactly $|L(e)|^2$ position variables, so $O(|E|M^2)$ in total. Each edge also needs exactly $|L(e)|$ constraints of the form given in Equation 4.1, and $|L(e)|$ constraints of the form given in Equation 4.2, so $O(|E|M)$ in total. There are $\binom{|L(e)|}{2}$ unique line pairs per edge, so we have to consider $O(M^2)$ line pairs for avoiding crossings per edge. If we don't require lines to follow simple paths, A and B might extend together into *multiple* edges, and might therefore cross multiple times in a single node. In total, we therefore need $O(|E|DM^2)$ crossing decision variables (both crossings and intra-path crossings are considered here), where D is the maximum node degree of the input line graph.

For Equations 4.3, 4.4, and 4.5, we have to consider all possible position pairs for two lines A and B on edges, so never more than $\binom{M}{2} < M^2$ per line pair and edge. For split crossings, we indeed only require at most $\binom{M}{2}$ constraints of the form given in Equation 4.3. For intra-path crossings and line separations, we have to consider all position pairs on both edges. Again, A and B might extend into *multiple* edges, so we need at most $\binom{M}{2}^D < M^{2D}$ per line pair, where D is the maximum input node degree. We hence need to add $O(|E|M^{2(1+D)})$ constraints to count crossings and separations. The total number of variables is therefore $O(|E|DM^2)$, and the total number of constraints is $O(|E|M^{2(1+D)})$.

4.8 Improved ILP Formulation

The exponential number of constraints in the baseline ILP formulation is due to explicitly enumerating all possible position assignments which induce a crossing, for each line pair. We would like to have a polynomial ILP size. Recall that we perform this enumeration to test whether $p_e^v(A) < p_e^v(B)$ and $p_f^v(A) < p_f^v(B)$. If we could efficiently check whether $p_e^v(A) < p_e^v(B)$, this would greatly reduce the number of constraints in our ILP.

4.8.1 Range Line Positions

Instead of using a binary decision variable x_{elp} which is 1 if line l is positioned at p on edge e, we introduce a variable $x_{el\leq p} \in \{0, 1\}$ to decide whether the position of line l is *smaller* than p on e. That is, if $x_{el\leq p} = 0$, $p > p$, and if $x_{el\leq p} = 1$, $p \leq p$. For a single line A positioned at P, we then have a sequence of such variables for each position slot $p = 1, \ldots, |L(e)|$, with $x_{el\leq p} = 1$ for all $p \leq P$, and $x_{el\leq p} = 0$ for all $p > P$. To ensure that a position is assigned to l, this sequence has to eventually arrive at 1, and only switch from 0 to 1 at most once. We enforce the latter with the following set of constraints:

$$\forall l \in L(e)\ \forall p \in \{1, \ldots, |L(e)| - 1\} : \quad x_{el\leq p} \leq x_{el\leq(p+1)}. \tag{4.6}$$

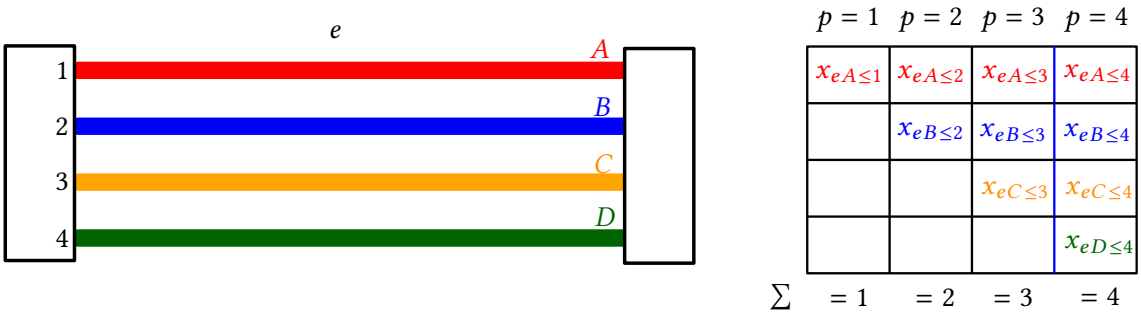


Figure 4.15: An edge e with 4 lines A, B, C, D assigned to positions 1, 2, 3, 4. The table shows the non-zero range line position variables in an assignment which satisfies Equations 4.6 and 4.6, thus ensuring a bijective mapping from positions to lines.

To ensure that each l is assigned a unique position, we also add the following constraints:

$$\forall p \in \{1, \ldots, |L(e)|\} : \quad \sum_{l \in L(e)} x_{el \leq p} = p. \tag{4.7}$$

So for each position slot p, the number of lines that claim to be positioned before or at p is exactly p. The constraint for case $p = 1$ therefore ensures that *exactly* one line is assigned to slot 1, the constraint for case $p = 2$ ensures that an additional different line is positioned at $p = 2$ (it cannot be the line from position 1 again, because then only one $x_{el \leq 2}$ would be 1), and so on. See Figure 4.15 for an example. As each position must thus be filled with a unique line, all lines get a position, and no position slot is overfilled.

4.8.2 Crossing Oracle

To avoid the enumeration of all crossing-inducing position pairs, we would now like to have variables $x_{eA<B}, x_{eB<A} \in \{1, 0\}$ for a line pair (A, B) telling is whether the position of line A is smaller than the position of line B on an edge e, or vice versa. This would enable us to directly check whether a crossing is induced by comparing the respective variables. Observe that the number of range position variables which are set to 1 directly indicates how small a position is - if A is positioned at $p = 1$, then all range position variables $x_{eA \leq p}$ will be 1 (see Figure 4.15, right). We can therefore ensure such an assignment with the following set of constraints:

$$\sum_{p=1}^{|L(e)|} x_{eA \leq p} - \sum_{p} x_{eB \leq p} + x_{eB<A} M \geq 0, \tag{4.8}$$

$$x_{eB<A} + x_{eA<B} = 1. \tag{4.9}$$

The sum difference $\sum_{p=1}^{|L(e)|} x_{eA \leq p} - \sum_p x_{eB \leq p}$ is exactly the (signed) position gap between A and B. If B is positioned before A, the difference of the sums is negative (it may never be smaller than $-M$), and the only way to fulfill the constraint is to set $x_{eB<A}M$ to 1. If on the other hand A is positioned before B, then the $x_{eA \leq p}$ sum is higher than the $x_{eB \leq p}$ sum, the difference is guaranteed to be positive, and $x_{eB \leq B}$ may be set to zero. To enforce that it is actually set to zero, Equation 4.9 makes sure that for a line pair (A, B), one of $x_{eB<A}$ and $x_{eA<B}$ is set to 1, and the other to zero.

Intra-Path Crossings

Our crossing oracle now enables us to set the intra-path crossing variable x_{efAB} fairly directly (assuming that the position in $e = \{u, v\}$ and $f = \{v, w\}$ are again both given w.r.t. v):

$$\left| x_{eA<B} - x_{fB<A} \right| - x_{efAB} \leq 0 \tag{4.10}$$

If the relative ordering of A and B is different in e than in f, then the absolute difference $|x_{eA<B} - x_{fB<A}|$ is 0, and x_{efAB} may remain 0, as indeed no crossing occurs. If their relative ordering w.r.t. v is the same, a crossing occurs, and the absolute value difference is 1, enforcing us to set x_{efAB} to 1. To model the absolute value computation in our ILP, we use the following standard replacement:

$$x_{eA<B} - x_{fB<A} - x_{efAB} \leq 0, \tag{4.11}$$

$$-x_{eA<B} + x_{fB<A} - x_{efAB} \leq 0. \tag{4.12}$$

If $x_{eA<B} = x_{fB<A}$, then x_{efAB} may still be zero as before. If $x_{eA<B} = 1$ and $x_{fB<A} = 0$, then Equation 4.11 will enforce $x_{efAB} = 1$. If $x_{eA<B} = 0$ and $x_{fB<A} = 1$, then Equation 4.12 will enforce $x_{efAB} = 1$.

Split Crossings

For split crossings where A and B extend into different edges f and g, we again have to only consider the relative ordering of A and B at e. Then a crossing either occurs if the position of A is smaller than B, or vice versa. Hence, we add the following set of constraints to get the correct assignment for the split crossing variable x_{eAB}. If a crossing would occur if A is positioned before B, we add the following constraint:

$$x_{eA<B} - x_{eAB} \leq 0. \tag{4.13}$$

If a crossing would occur if A is positioned after B, we add the following constraint:

$$x_{eB<A} - x_{eAB} \leq 0. \tag{4.14}$$

4.8.3 Line Separations

Just as for our baseline ILP, we would like to have a binary decision variable $x_{eA\|B} \in \{0,1\}$ which is 0 if $\left|p_e^v(A) - p_e^v(B)\right| = 1$ (if A and B are placed next to each other) and 1 otherwise, but without enumerating all separation-inducing position pairs like we did in Section 4.7.3. A separation may only occur if A and B extend from an edge e into an edge f. Obviously, if $|L(e)| = |L(f)| = 2$, A and B can never separate, and we do not need explicit separation variables. We could then get the desired assignment with the following set of constraints for each line pair $\{A, B\}$ on an edge e:

$$\left|\sum_{p=1}^{|L(e)|} x_{eA\leq p} - \sum_{p} x_{eB\leq p}\right| - x_{eA\|B} M \leq 1. \tag{4.15}$$

Recall that $\sum_{p=1}^{|L(e)|} x_{eA\leq p} - \sum_p x_{eB\leq p}$ is the signed position gap of A and B, the absolute value of which is 1 exactly then if they are placed next to each other. Again, we use the following replacement for the absolute value computation:

$$\sum_{p=1}^{|L(e)|} x_{eA\leq p} - \sum_{p} x_{eB\leq p} - x_{eA\|B} M \leq 1, \tag{4.16}$$

$$\sum_{p=1}^{|L(e)|} x_{eB\leq p} - \sum_{p} x_{eA\leq p} - x_{eA\|B} M \leq 1. \tag{4.17}$$

Note, however, that $x_{eA\|B}$ might always be set to 1. To ensure that it is indeed only set to 1 if A and B are not placed next to each other, we additionally add the following single constraint per edge e:

$$\sum_{(l,l')\in U(e)} x_{el\|l'} \leq \binom{|L(e)|}{2} - |L(e)| - 1. \tag{4.18}$$

There are indeed $\binom{|L(e)|}{2}$ unique line pairs on $L(e)$, and exactly $|L(e)| - 1$ of them will be placed next to each other. So exactly $\binom{|L(e)|}{2} - |L(e)| - 1$ lines are separated on e, and this number is already exactly matched by the sum of variables $x_{eA\|B}$ which are forced to have value 1.

Like for crossings, we now want a binary decision variable $x_{efA\|B} \in \{0,1\}$ which should be 1 if A and B separate while extending from e to f, or 0 otherwise. Again, the desired assignment would be satisfied with an absolute value constraint of the form

$$|x_{eA\|B} - x_{fA\|B}| - x_{efA\|B} \leq 0, \tag{4.19}$$

which we again model as follows:

$$x_{eA\|B} - x_{fA\|B} - x_{efA\|B} \leq 0, \tag{4.20}$$

$$-x_{eA\|B} + x_{fA\|B} - x_{efA\|B} \leq 0. \tag{4.21}$$

Note that if A and B extend from e to f, but $|L(e)| = 2$, the following constraint is enough:

$$x_{fA\|B} - x_{efA\|B} = 0. \tag{4.22}$$

The decision variable $x_{efA\|B}$ is then added to the objective function with its weight reflecting the separation penalty between A and B on node v connecting edges e and f.

4.8.4 Improved ILP Size

For each line on each edge, we have $|L(e)|$ range line position variables, so no more than $|E|M^2$ in total.

For the crossing oracle, we add $O(|E|M^2)$ variables of the form $x_{eB<A}$, and asymptotically just as many constraints (Equations 4.8 and 4.9) to ensure their correct assignment. We then again add $O(|E|DM^2)$ split- and intra-path crossing variables, and asymptotically just as many constraints again (Equations 4.12, 4.11, 4.11, 4.13, and 4.14).

For the line separations, we again add at most $|E|M^2$ variables of the type $x_{eA||B}$ and just as many constraints to ensure their desired assignments. Finally, we add $O(|E|DM^2)$ line separations variables of the type $x_{efA||B}$, and again just as many constraints for them. In total, we now have $O(|E|DM^2)$ variables and $O(|E|DM^2)$ constraints. As desired, the ILP size is polynomial now.

4.9 Line Graph Simplification

Real-world input line graphs typically contain segments or paths for which at least a partial optimal ordering of the lines is intuitively obvious (see Figure 4.41, top, for an example). This section aims to formalize this intuition into provable lemmas. We then derive several simplification rules that may be applied to the input line graph G in polynomial time prior to finding optimal line orderings. These simplifications rules do not affect the global line ordering optimality. Generally, this means that we can easily construct an optimal line ordering solution σ on G from an optimal line ordering solution σ' on the simplified graph G'. The simplification rules described in this section have the potential to greatly reduce the overall search space size. Additionally, they may break up the input line graphs into multiple connected components, which may then be solved by appropriate methods (either the ILP described above, a heuristic, or even an exhaustive search), possibly in parallel.

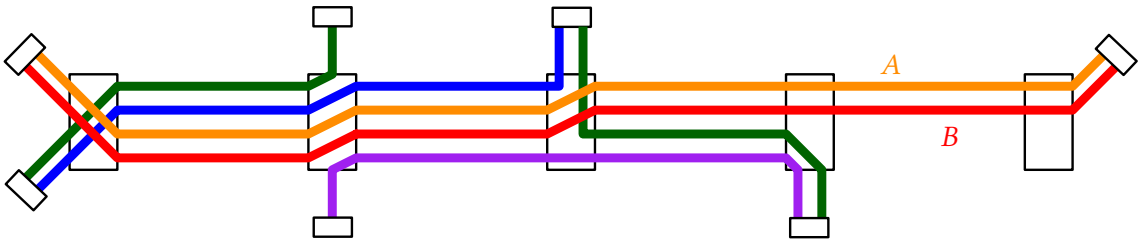


Figure 4.16: If two lines A and B strictly follow the same simple path and have the same crossing and separation weights at each node, an optimal placement will always be to put them next to each other, crossing-free, in each edge. This might not be the only optimal placement, however.

This Section is organized as follows: We first prove Lemmas 4.11 - 4.15. We then use them to derive a set of pruning (Section 4.9.2), cutting (Section 4.9.3) and untangling rules (Section 4.9.4). For each rule, we prove that the transformation does not affect global optimality. We will then discuss the complexity of a full graph simplification in Section 4.9.6, and describe several instances for which the simplification rules completely solve the corresponding MLNCM problem (Section 4.10).

4.9.1 General Observations

We consider Figure 4.16. It is not hard to see that if two lines follow the same simple path through G, an optimal ordering is to always place them next to each other. However, an optimal line ordering solution will not necessarily place them this way, see Figure 4.9, right for a crossing- and separation-free counter-example.

Lemma 4.11. *Given a set of lines $\mathcal{B} = \{A, B, C, \dots\} \subseteq \mathcal{L}$ which all strictly follow the same simple path P (lines are not allowed to branch). Additionally, we require crossing and separation weights for all $l \in \mathcal{B}$ to be equal in all path nodes. For any ordering $l_1, \dots, l_{|\mathcal{B}|}$ on the $\mathcal{B}$ a globally optimal line ordering solution exists in which for any two adjacent path edges $e = \{u, v\}$, $f = \{v, w\}$ and $j > 1$ it holds that $p_e^v(l_j) = p_f^w(l_j) = p_e^v(l_{j-1}) + 1$ (all $l \in \mathcal{B}$ are always positioned next to each other with the same ordering, free of internal crossings or separations).*

Proof. Let σ be a line ordering solution for which the property does not hold, and let $m \in \mathcal{B}$ be the line in $\mathcal{B}$ that induces the lowest cost (sum of weighted line separations and crossings). We claim that we may place any $l \in \mathcal{B} \setminus \{m\}$ next to m on the path at lower or equal cost. Obviously the new position will not introduce any crossings or separations between l and m. All separations that may be induced by l at the new position were already induced by m. The weighted crossings l will induce at the new position will be equal to the weighted crossings m induces. As the total cost induced by m was smaller than the total cost induced by l, the new solution σ' will hence always be

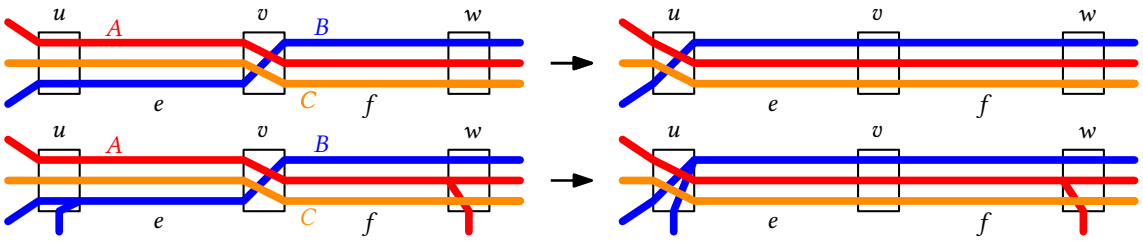


Figure 4.17: Top: If all lines from e distinctly extend over u, and if all crossing and separation weights at u are cheaper than or equal to those at v, we can base the line ordering of e on that of f, moving each crossing and separation from u to v. Bottom: If lines do not distinctly extend over u, moving the crossings to u might now result in multiplied crossing costs for lines which branch at u.

better than or equal to σ. We can thus produce any permutation of $\mathcal{B}$ with all lines next to each other, and it will be an optimal placement of the $\mathcal{B}$. □

The following lemma is similar to Lemma 4.2, but also considers crossing and separation weights:

Lemma 4.12. *Given a line graph $G = (V, E, L, \mathcal{L})$, a line ordering solution σ, and edges $e = \{u, v\}$ and $f = \{v, w\}$, with $L(e) = L(f)$ and $\deg(v) = 2$. If all lines $L(e)$ either end or uniquely extend over u, and if in the latter case crossing and separation weights between all lines in $L(e)$ are equal or lower at u than at v, we can make v crossing and separation free without affecting optimality.*

Proof. If we set $p_e^v = p_f^w$, v will be crossing and separation free, and all crossings or separations that previously occurred at v now occur at u (some may not occur at all if a line partner has a terminus at u, in which case σ was not optimal). As all crossing and separation weights are equal to or cheaper to those at u, the thus obtained ordering solution σ' will be equal or better than σ. □

Observe that if a line does not distinctly extend over u, crossing costs might be multiplied there, which is why we excluded such cases. Figure 4.17 illustrates this.

Lemma 4.13. *Given an edge $e = \{u, v\}$ where each $l \in L(e)$ has terminates at u. The ordering of $L(e)$ will not affect the number of crossings or separations in u.*

Proof. No $l \in L(e)$ extend over v, so they cannot induce any crossing or separation (Figure 4.18). □

Lemma 4.14. *Given an edge $e = \{u, v\}$ with $|L(e)| = 1$. The ordering of $L(e)$ will not affect the global number of crossings or separations.*

Proof. All orderings of $L(e)$ are equivalent, as there is only one. □

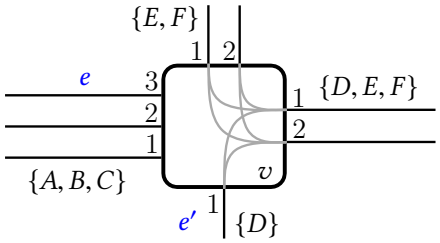


Figure 4.18: Illustration of Lemma 4.13.

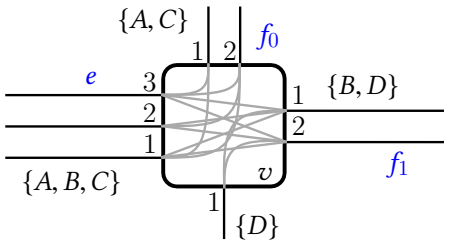


Figure 4.19: Illustration of Lemma 4.15.

We now consider split crossings as defined in Section 4.1.1. The following lemma directly follows from the definition:

Lemma 4.15. *If all lines $L(e)$ for an edge $e = \{u, v\}$ uniquely extend over v into n edges $f_i \neq e$, the circular edge ordering of the f_i induces a partial ordering on the $L(e)$ which does not induce any crossings or separations at v between two lines $l, l' \in L(e)$ which continue into different edges f_1 and f_2, namely a partial ordering which ensures $\overleftarrow{p_e^v}(l) > \overleftarrow{p_e^v}(l')$ if $\pi_e^v(f_1) > \pi_e^v(f_2)$.*

Proof. Per definition, a separation may only occur between lines continuing together from an edge e into and edge f, so the relative ordering of such line bundles has no effect on the number of separations at v. As l and l' extend into different edges, only a split crossing may occur, but per definition of a split crossing, the partial ordering can never induce it (Figure 4.19). □

4.9.2 Pruning Rules

This section describes several simple rules to prune to line graph G prior to the line ordering optimization, provably without affecting the optimality of the line orderings found on the simplified graph G'.

In Sections 4.9.3 and 4.9.4, we will describe rules that split line graph nodes. We therefore first define v^* as the original input line graph node of v (which may be v itself). If any of the following rules splits a node v into nodes v' and v'', the crossing and separation penalties at v' and v'' will be set to the crossing and separation penalties at the original node v.

Pruning Rule 1 (Node Contraction)**.** *Contract each node v with $\deg(v) = 2$ with adjacent edges $e = \{u, v\}, f = \{v, w\}$ if the following holds:*

1. *$L(e) = L(f)$*
2. *$\forall l_1, l_2 \in L(e)^2, l_1 \neq l_2$: All lines terminate or have lower crossing and separation weights at u, or all lines terminate or have lower crossing and separation weights at v.*

Figure 4.20 provides an example.

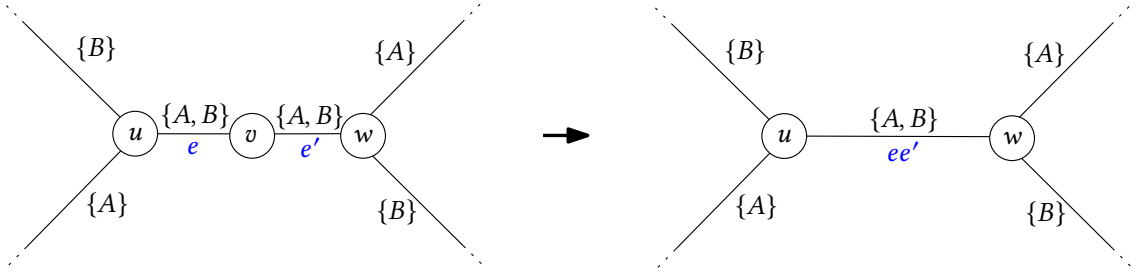


Figure 4.20: Illustration of Pruning Rule 1 (Node Contraction). Node v is adjacent to two edges e and e' holding the same lines ($L(e) = L(e')$). We can contract v if the crossing and separation costs at u or w are cheaper than or equal to those at v without affecting optimality.

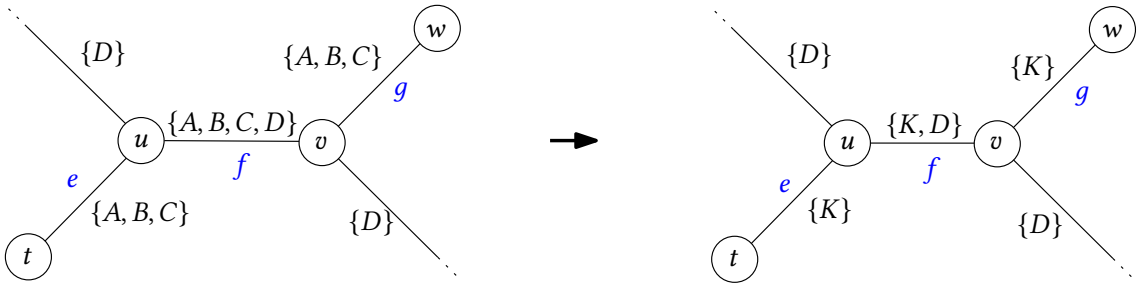


Figure 4.21: Illustration of Pruning Rule 2 (Line Partner Collapse). If a group $\mathcal{B}$ of lines (here, $\mathcal{B} = \{A, B, C\}$) follows the same simple path, we can replace them by a single line (here: K) without affecting optimality if the crossing weights for the new line are set to the sum of the crossing weights of each $l \in \mathcal{B}$ are each affected node.

Lemma 4.16. *Pruning Rule 1 (Node Contraction) does not affect the optimality of the line ordering.*

Proof. Lemma 4.12 holds for v. Any optimal solution σ with crossings or separations at v can thus be transformed into an optimal solution σ' where v is crossing and separation free. We therefore do not have to consider v during optimization. □

Pruning Rule 2 (Line Partner Collapse)**.** *Replace a set $\mathcal{B} \subseteq \mathcal{L}$ of lines with a single new line k if the following holds:*

1. *All lines $l \in \mathcal{B}$ follow the exact same simple path P through the line graph.*
2. *For all $l \in \mathcal{B}$ and at each path nodes, crossing and separation weights are equal.*

Set all crossing weights for k to be the sum of the respective weights for all $l \in \mathcal{B}$.

Figure 4.21 provides an example.

Lemma 4.17. *Pruning Rule 2 (Line Partner Collapse) does not affect the optimality of the line ordering.*

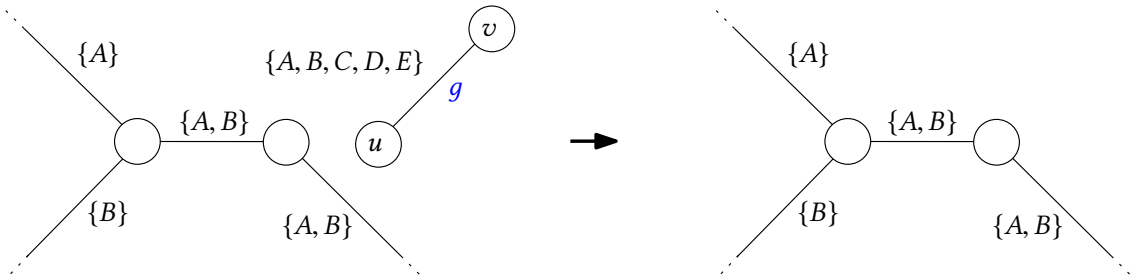


Figure 4.22: Illustration of Pruning Rule 3 (Single Edge Prune).

Proof. Lemma 4.11 holds, and there will always be an optimal line ordering solution in which the lines $\mathcal{B}$ are placed next to each other with the same ordering in all path edges. We can thus settle this ordering at random initially, optimize with the placeholder line k, and insert the initial ordering into the optimal line ordering without affecting optimality. □

Pruning Rule 3 (Single Edge Prune)**.** *Delete edges* $e = \{u, v\}$ *with* $\deg(u) = \deg(v) = 1$.

Figure 4.22 provides an example.

Lemma 4.18. *Pruning Rule 3 (Single Edge Prune) does not affect the optimality of the line ordering.*

Proof. Per Lemma 4.13, the ordering of $L(e)$ will not affect the number of crossings or separations in any solution. □

4.9.3 Cutting Rules

While the pruning rules may already greatly reduce the search space size of the line ordering optimization problem, we would like to partition the input line graph into components which can be optimized separately and in parallel. We call a line graph component whose line orderings does not affect the number of crossings or separations in the rest of the line graph an ordering-relevant connected component. In this section, we describe several cutting rules which transform the input line graph in such a way that any ordering-relevant connected component in the original line graph has a corresponding connected component in the transformed graph.

Like in Section 4.9.2, we show for each cutting rule that it will not affect the optimality of the final line ordering.

Cutting Rule 1 (Single Line Cut)**.** *Cut an edge* $e = \{u, v\}$ *with* $\deg(u) > 1$ *and* $\deg(v) > 1$ *into two edges* $e' = \{u, v'\}$ *and* $e'' = \{v'', v\}$ *if* $|L(e)| = 1$. v' *and* v'' *are new nodes positioned somewhere along the original edge* e. *Set* $L(e') = L(e'') = L(e)$.

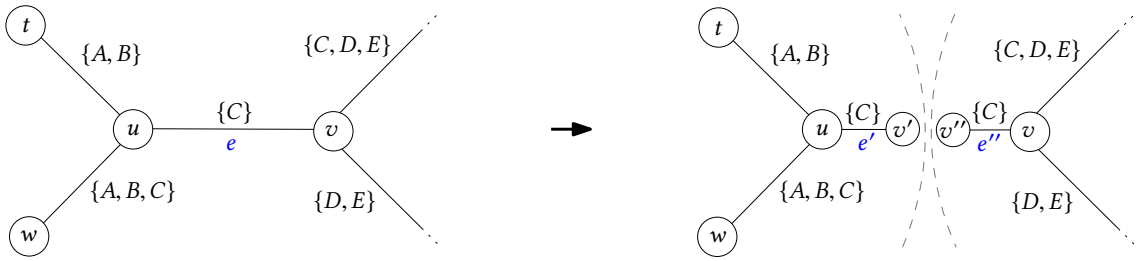


Figure 4.23: Illustration of Cutting Rule 1 (Single Line Cut). An edge e with $|L(e)| = 1$ may be cut as depicted without affecting optimality.

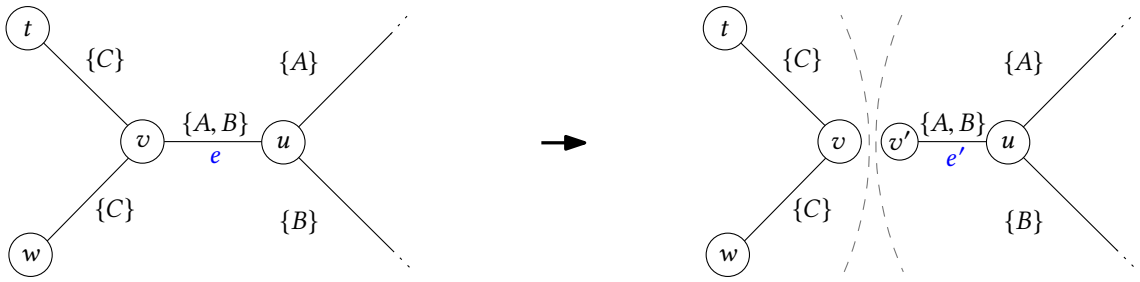


Figure 4.24: Illustration of Cutting Rule 2 (Terminus Detachment).

Figure 4.23 gives an example.

Lemma 4.19. *Cutting Rule 1 (Single Line Cut) does not affect the optimality of the line ordering.*

Proof. Per Lemma 4.13 the ordering of $L(e)$ is irrelevant for the number of crossings or separations. As $\deg(v') = \deg(v'') = 1$, no additional crossing or separation may occur in these new nodes. As $L(e') = L(e'') = L(e)$, the number of crossings and separations at v and u is also not affected. □

Cutting Rule 2 (Terminus Detachment)**.** *Given a node v adjacent to an edge $e = \{u, v\}$ for which the following holds:*

1. *v is a terminus station for all lines $l \in L(e)$.*
2. $\deg(v) > 1$.

Detach e from v. (Delete e, add a new node v' positioned somewhere along the original edge e, and add a new edge $e' = \{u, v'\}$ with $L(e') = L(e)$).

Lemma 4.20. *Cutting Rule 2 (Terminus Detachment) does not affect the optimality of the line ordering.*

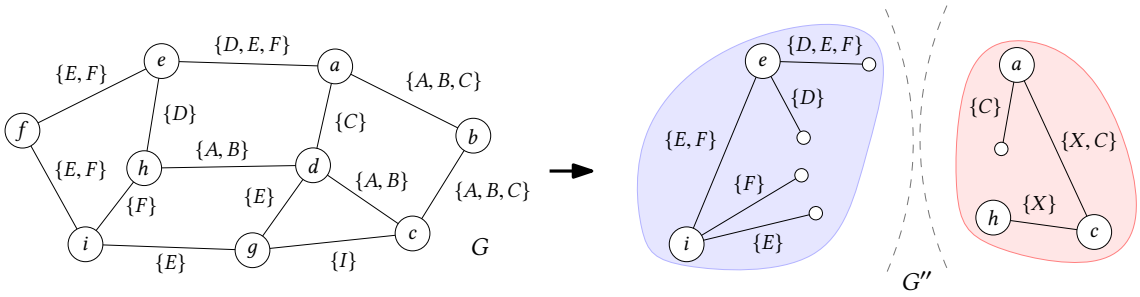


Figure 4.25: Left: Line graph with 7 lines A, B, C, D, E, F, G. Right: The graph G' resulting from the repeated application of pruning rules 1-3 and cutting rules 1-2 to G.

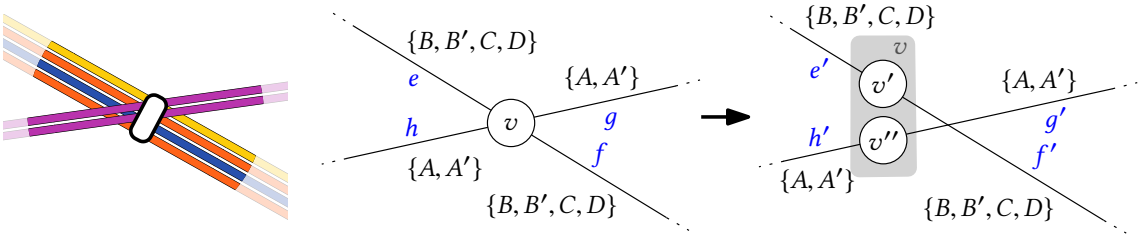


Figure 4.26: Left: Excerpt from the New York subway map. Middle: Corresponding line graph. Right: Line graph (exact edge geometries have been omitted) after Untangling Rule 1 was applied.

Proof. Per Lemma 4.13 the ordering of $L(e)$ is irrelevant for the number of crossings or separations in v. As $\deg(v') = 1$, no additional crossing or separation may occur in these new nodes. As $L(e') = L(e)$, the number of crossings and separations at u is also unaffected. □

4.9.4 Untangling Rules

We now consider Figure 4.26 and 4.27. In Figure 4.26, it is obvious that the line orderings on h and g can be optimized independently of the line orderings on e and f, but none of the pruning and cutting rules apply, as the two line bundles $\{A, A'\}$ and $\{B, B', C, D\}$ cross each other at v. In Figure 4.27, the situation is slightly more complex, as the crossing of two line bundles happens over three nodes u, v and w, but it is again intuitively obvious that the ordering of the line bundles can be optimized independently here. This section introduces several so-called untangling rules which aim to formalize these intuitions into formal simplification rules. The name untangling was chosen because in the best case, these rules "untangle" the line graph until only simple "threads" (with a line-ordering search space size of 1) remain. Again, each untangling rule is proven to not affect line ordering optimality.

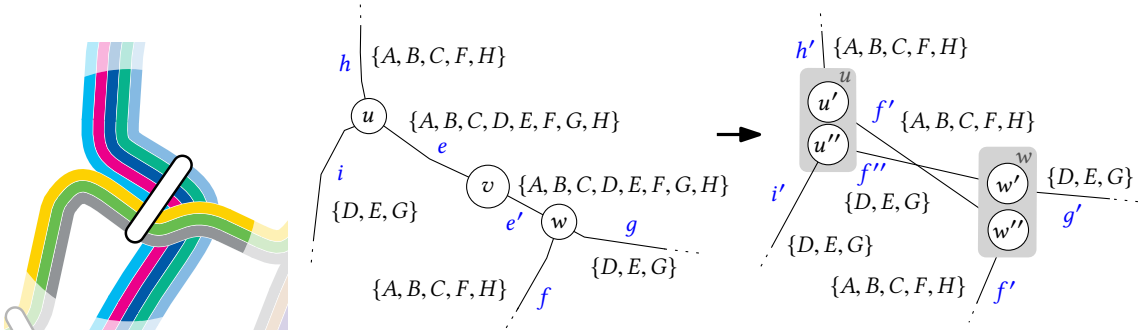


Figure 4.27: Left: Excerpt from the Stuttgart light rail map. Middle: Corresponding line graph. Right: Line graph (exact edge geometries have been omitted) after Pruning Rule 1 and Untangling Rule 3 was applied. The number of possible line orderings on e and e' was reduced from $|L(e)|! \cdot |L(e')|! = 8! \cdot 8! = 1.63 \cdot 10^9$ to $|L(f')|! \cdot |L(f'')|! = 5! \cdot 3! = 720$.

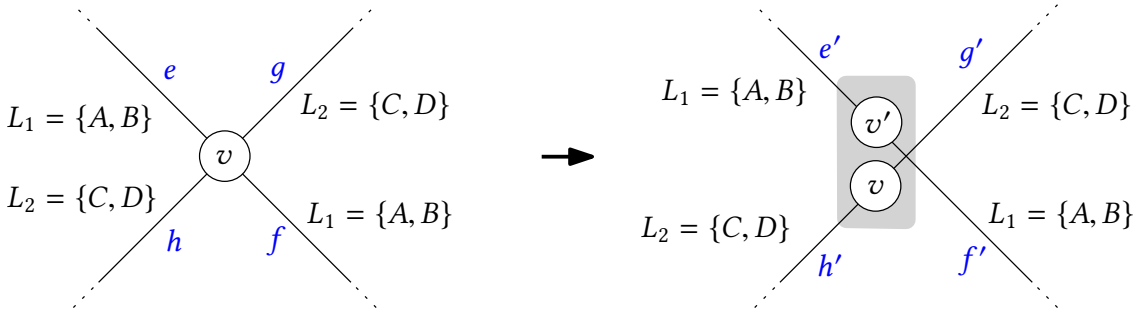


Figure 4.28: Illustration of Untangling Rule 1. Left: Two line bundles L_1 and L_2, with $L_1 \cap L_2 = \emptyset$ cross at a node v. The ordering of L_1 on h and g can be optimized independently of the ordering of L_2 on e and f. Right: Untangling Rule 1 has been applied to v.

Full X Structure

We first consider situations as depicted in Figure 4.28, left, which we call a *Full X* structure, and define the following untangling rule on them:

Untangling Rule 1 (Full X). *Given a node v with* $\deg(v) > 2$ *and with $n > 2$ adjacent edges with at least one line for which v is* not *a terminus (to remove trivial cases that can be transformed into cases eligible for Pruning Rule 1 (Node Contraction)).*

We consider all pairs $\{e, f\}$ of adjacent edges $e = \{v, u\}$ and $f = \{v, w\}$. If for such a pair it holds that $L(e) = L(f)$, and if all $l \in L(e)$ uniquely extend over v, delete e and f, add a new node v' at the same position as v, and add new edges $e' = \{v', u\}$ and $f' = \{v', w\}$ with $L(e') = L(f') = L(e)$.

Figure 4.28, right gives an example.

Lemma 4.21. *Untangling Rule 1 does not affect the optimality of the line ordering.*

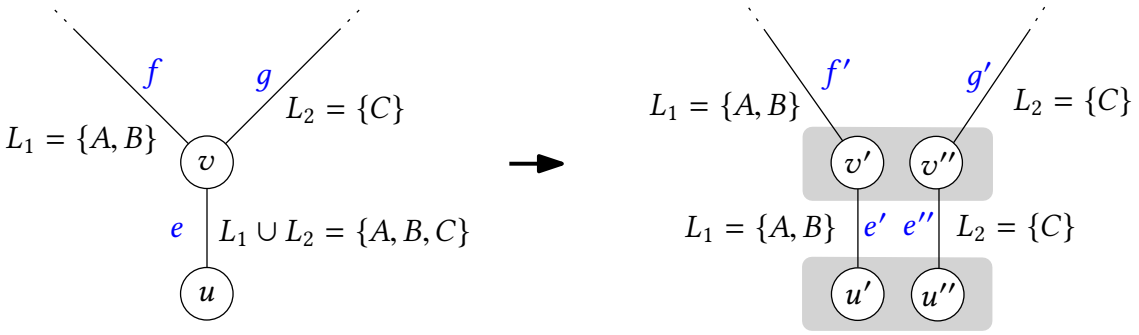


Figure 4.29: Illustration of Untangling Rule 2. If a major leg (here edge e) terminates at node u and branches at node v into two minor legs (here edges f and g), v and e can be split like depicted.

Proof. No crossing or separation between any line $l \in L(e)$ and any line $m \notin L(e)$ was possible originally. All crossings or separations between lines $l \notin L(e)$ may still occur in v. All crossings or separations between lines $l \in L(e)$ that may have occurred in v can still occur in v'. □

This rule has two important effects: First, the nodes v and v' may now be eligible for contraction using Pruning Rule 1 (Node Contraction). Second, v and v' may now be in different connected components which can be optimized separately.

Single Y Structures

Figure 4.29 depicts a situation in which two distinct line bundles L_1 and L_2 join at a node v and continue over a single edge e to a node u, which is a terminus for all $l \in L_1 \cup L_2$. It is intuitively obvious that any crossing or separation between a line $l \in L_1$, and a line $m \in L_2$ at node v can be avoided, as for any line ordering on f and g, there is line ordering for e which does not induce such a crossing or separation. This intuition is formalized in the following untangling rule:

Untangling Rule 2 (Full Y). *Given a node v adjacent to an edge $e = \{u, v\}$ with $\deg(u) = 1$ and the following additional properties:*

1. *Each $l \in L(e)$ terminates at u.*
2. *Each $l \in L(e)$ uniquely extends over v into one of $\deg(v) - 1$ edges $e_1, \ldots, e_{\deg(v)}$, with $e_i \neq e$, and all $l \in L(e_i)$ uniquely extend over v into e (that is, $L(e_1) \cup \ldots \cup L(e_{\deg(v)-1}) = L(e)$ and $L(e_1) \cap \ldots \cap L(e_{\deg(v)-1}) = \emptyset$).*

We call e the major leg, and $e_1, \ldots, e_{\deg(v)}$ the minor legs. Split u and v into nodes u', u'' and v', v''. Connect v' and u' with an edge e' and set $L(e') = L(e_1)$ (the lines of the leftmost minor leg). Connect v'' and u'' with an edge e'' and set $L(e'') = \bigcup_{i=2}^{\deg(v)-1} L(e_i)$ (the lines of the remaining minor legs to the right). Connect v' to the node that v was originally

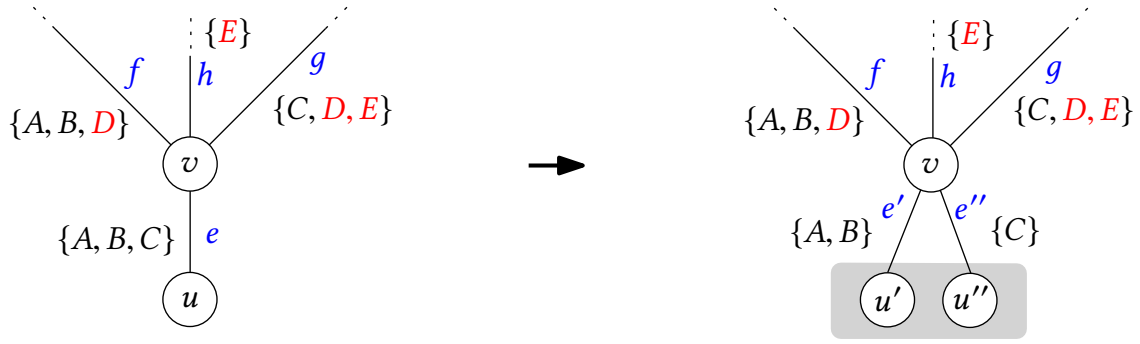


Figure 4.30: Illustration of Untangling Rule 3.

connected by e_1 with a new edge e'_1. Set $L(e_1) = L(e'_1)$. Connect all remaining minor legs in a similar way to v''

Note that this rule does not only consider Y structures with two minor legs (as depicted in Figure 4.29), but works with any number of minor legs.

Lemma 4.22. *Untangling Rule 2 does not affect the optimality of the line ordering.*

Proof. Per Lemma 4.15, the minor legs induce a partial ordering on the lines on e which does not induce any crossing or separation between two lines coming from different minor legs. Splitting v therefore does not prevent any optimal line crossings or separations at v between lines from different minor legs. Per Lemma 4.13, the line ordering on e has no effect on the number of crossings or separations in u, and as $\deg(u) = 1$, there cannot be any crossings or separations in u. Splitting u therefore does also not prevent any optimal line crossings or separations at u. Any crossing or separation at v between a line on a minor leg, and a line on the major leg is still possible in the corresponding split node. □

To be later able to reconstruct an ordering of the original edge e from the optimal ordering on edges e' and e'', we have to store an ordering of e' and e'' with respect to e. This is just the original clockwise ordering between e_1 and the remainder of the minor legs. If Untangling Rule 2 (or any of the untangling rules defined below) is applied repeatedly, we end up with a hierarchy of edge orderings, from which (given optimal line orderings for the leafs of this tree) the line ordering for the original input graph must be reconstructed.

A slightly different kind of Y-structure, which we call a *partial* Y, is depicted in Figure 4.30. Just like in the previous full Y-structure, the lines on the major leg e completely and uniquely branch into the minor legs f and g. However, the lines on the minor legs are not pairwise disjoint: a line D is present on f and g. While Lemma 4.15 still holds for e, we cannot simply split v into distinct nodes per minor legs as before, as separations

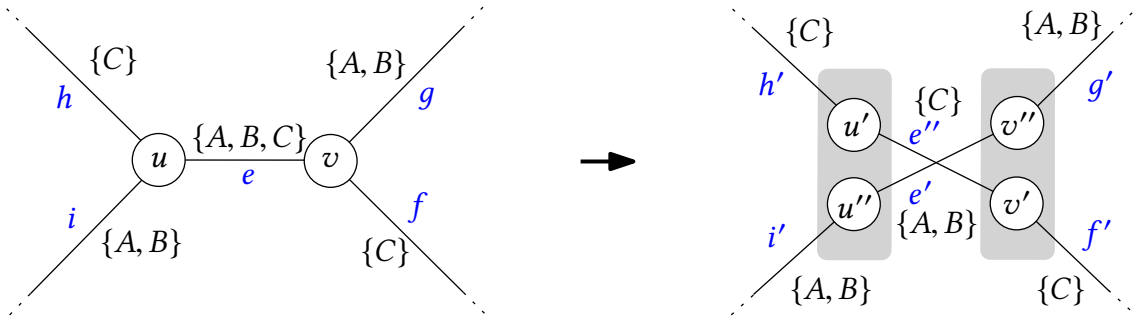


Figure 4.31: Illustration of Untangling Rule 4, with inverse minor legs.

or crossings involving D may occur there. We may still split u, however, as captured in the following rule:

Untangling Rule 3 (Partial Y)**.** *Given a node v adjacent to an edge $e = \{u, v\}$ with* $\deg(u) = 1$ *and the following additional properties:*

1. *Each $l \in L(e)$ terminates at u.*
2. *Each $l \in L(e)$ uniquely extend over v into one of $n > 1$ edges $e_1, \ldots, e_n, e_i \neq e$, and each e_i has a line that uniquely extends into e (that is, $L(e_1) \cup \ldots \cup L(e_n) \supseteq L(e)$, $L(e_1) \cap \ldots \cap L(e_n) \cap L(e_n) = \emptyset$ and $\forall e_i : L(e_i) \cap L(e) \neq \emptyset$).*

We again call e the major leg, and $e_1, \ldots, e_n$ the minor legs. Split u into nodes u', u''. Connect v and u' with an edge e' and set $L(e') = L(e_1)$ (the lines of the leftmost minor leg). Connect v and u'' with an edge e'' and set $L(e'') = \bigcup_{i=2}^{n-1} L(e_i)$ (the lines of the remaining minor legs to the right).

Lemma 4.23. *Untangling Rule 3 does not affect the optimality of the line ordering.*

Proof. Lemma 4.15 and Lemma 4.13 still hold as in the previous proof, we may therefore split e in the described fashion without affecting optimality. □

Note that Untangling Rule 2 only detaches a single minor leg, although the prerequisites allow for an arbitrary number of minor legs. To untangle such structures, a repeated application of the untangling rules as described above is necessary.

Double Y Structures

We now consider Figure 4.31, left, which is a condensed version of the real-world example in the Stuttgart light rail network given in Figure 4.27. We call such structures *Double Y* and define the following untangling rule on them:

Untangling Rule 4 (Full Double Y)**.** *Given an edge $e = \{u, v\}$ with* $\deg(u) = \deg(v) \geq 3$ *and the following additional properties:*

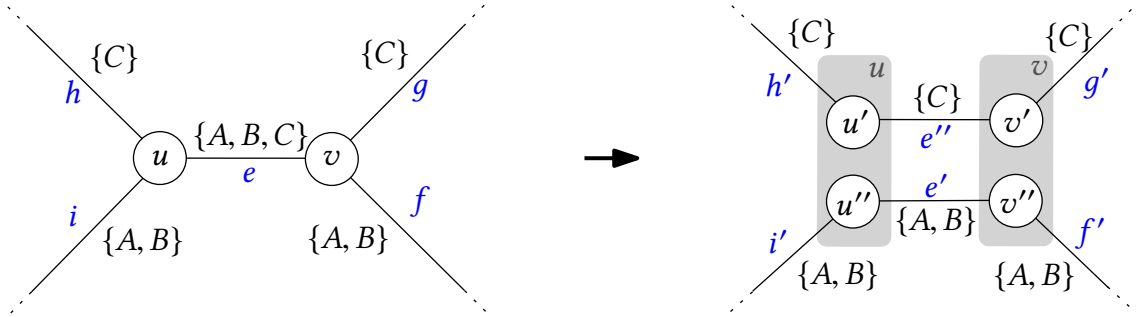


Figure 4.32: Illustration of Untangling Rule 4, with matching minor legs.

1. *Each* $l \in L(e)$ *uniquely extends over* v *into one of* $\deg(v)-1$ *edges* $f_1, \ldots, f_{\deg(v)}$, $f_i \neq e$, *and all* $l \in L(f_i)$ *uniquely extend over* v *into* e *(that is,* $L(f_1) \cup \ldots \cup L(f_{\deg(v)-1}) = L(e)$ *and* $L(f_1) \cap \ldots \cap L(f_{\deg(v)-1}) = \emptyset$*).*
2. *Each* $l \in L(e)$ *uniquely extends over* u *into one of* $\deg(u)-1$ *edges* $g_1, \ldots, g_{\deg(v)}$, $g_i \neq e$, *and all* $l \in L(g_i)$ *uniquely extend over* v *into* e *(that is,* $L(g_1) \cup \ldots \cup L(g_{\deg(v)-1}) = L(e)$ *and* $L(g_1) \cap \ldots \cap L(g_{\deg(v)-1}) = \emptyset$*).*
3. *There is a bijective mapping* $A : \{f_1, \ldots, f_{\deg(v)}\} \mapsto \{g_1, \ldots, g_{\deg(u)}\}$ *such that* $L(f_i) = L(A(f_i))$.
4. *Either* $\forall l_1, l_2 \in L(e)^2, l_1 \neq l_2 : w_\times(l_1, l_2, v) \leq w_\times(l_1, l_2, u)$, *or* $\forall l_1, l_2 \in L(e)^2, l_1 \neq l_2 : w_\times(l_1, l_2, u) \leq w_\times(l_1, l_2, v)$

We call e *the major leg,* f_i *the left minor legs, and* g_i *the right minor legs. Split* u *and* v *in the same fashion as in Untangling Rule 2, but ensure that* u' *and* v'*, as well as* u'' *and* v'' *are adjacent to matching minor legs.*

See Figure 4.31, right for an example.

We again have to store an ordering of e' and e'' to later be able to reconstruct a line ordering solution for the original input graph. Again, these orderings directly correspond to the partial ordering of $L(e)$ induced by either the minor legs at u, or the minor legs at v. But these partial orderings may not be compatible. If that was the case, the splitting of e hides unavoidable crossings that will be later re-introduced during the reconstruction of the line ordering for the original input graph. To ensure that these crossings will occur at the optimal position, we base the ordering of e' and e'' on the partial ordering induced by the minor legs at v if $\forall l_1, l_2 \in L(e)^2, l_1 \neq l_2 : w_\times(l_1, l_2, v) \leq w_\times(l_1, l_2, u)$, or on the partial ordering induced by the minor legs at v if otherwise.

Lemma 4.24. *Untangling Rule 4 does not affect the optimality of the line ordering.*

Proof. W.l.o.g., we assume $\forall l_1, l_2 \in L(e)^2, l_1 \neq l_2 : w_\times(l_1, l_2, u) \leq w_\times(l_1, l_2, v)$. Per Lemma 4.15, the minor legs at u induce a partial ordering on the $L(e)$ which does not induce any crossings or separations at u. Similarly, the minor legs at v induce a partial

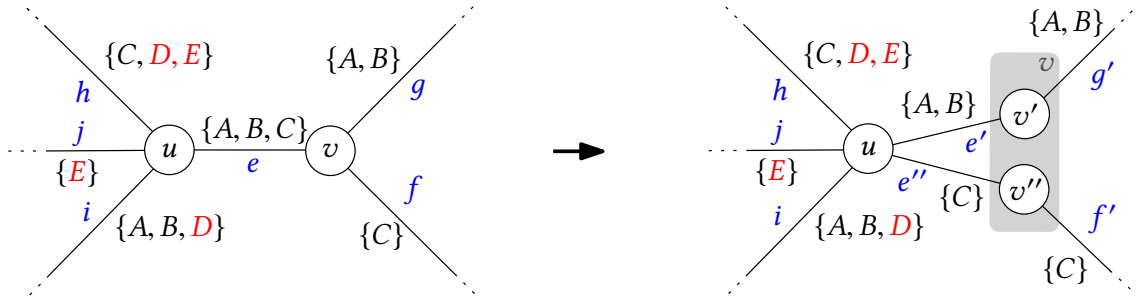


Figure 4.33: Illustration of Untangling Rule 5.

ordering on the $L(e)$ which does not induce any crossings or separations at v. There are now two cases: (1) The ordering induced at u is exactly inverse to the ordering induced at v (see Figure 4.32, left for an example). (2) The ordering induced at u is not inverse to the ordering induced at v (see Figure 4.31, left for an example). In case (1), the two orderings match, and any optimal line ordering solution will have no crossings or separations between two lines l from different minor legs, and no optimal crossing or separation is prohibited by Untangling Rule 4. In case (2), the induced partial ordering will also not add any separations at u or v between two lines from different minor legs, as no two such lines continue together from e over u or v. The inter-partion crossings induced in case (2) are unavoidable and must occur either at u, or at v. Implicitly moving them to v by an appropriate ordering of the split edges e' and e'' is optimal, as we assumed $\forall l_1, l_2 \in L(e)^2, l_1 \neq l_2 : w_\times(l_1, l_2, u) \leq w_\times(l_1, l_2, v)$. □

Figure 4.33, left shows the Double-Y equivalent of a partial Y structure, a *partial Double-Y* structure, which may be untangled in a similar way as partial Y structures:

Untangling Rule 5 (Partial Double Y)**.** *Given an edge* $e = \{u, v\}$ *with* $\deg(u) \geq \deg(v) \geq 3$ *and the following additional properties:*

1. *Each* $l \in L(e)$ *uniquely extends over* v *into one of* $n < \deg(v)$ *edges* $f_1, \ldots, f_n$, $f_i \neq e$, *and all* $l \in L(f_i) \cap L(e)$ *uniquely extend over* u *into* e *(that is,* $L(f_1) \cup \ldots \cup L(f_n) = L(e)$ *and* $L(f_1) \cap \ldots \cap L(f_n) \cap L(e) = \emptyset$*).*
2. *Each* $l \in L(e)$ *uniquely extends over* u *into one of* $\deg(u) - 1$ *edges* $g_1, \ldots, g_{\deg(v)}$, $g_i \neq e$, *and all* $l \in L(g_i)$ *uniquely extend over* v *into* e *(that is,* $L(g_1) \cup \ldots \cup L(g_{\deg(v)-1}) = L(e)$ *and* $L(g_1) \cap \ldots \cap L(g_{\deg(v)-1}) = \emptyset$*).*
3. *There is an injective mapping* $A : \{g_1, \ldots, g_{\deg(u)}\} \mapsto \{f_1, \ldots, f_{\deg(v)}\}$ *such that* $L(g_i) \subseteq L(A(g_i))$.
4. $\forall l_1, l_2 \in L(e)^2, l_1 \neq l_2 : w_\times(l_1, l_2, v) \leq w_\times(l_1, l_2, u)$.

Split v *in the same fashion as in Untangling Rule 4, but leave* u *as is.*

Note that the only difference to Untangling Rule 5 is that we allow additional edges at v, as well as lines extending over v from minor legs.

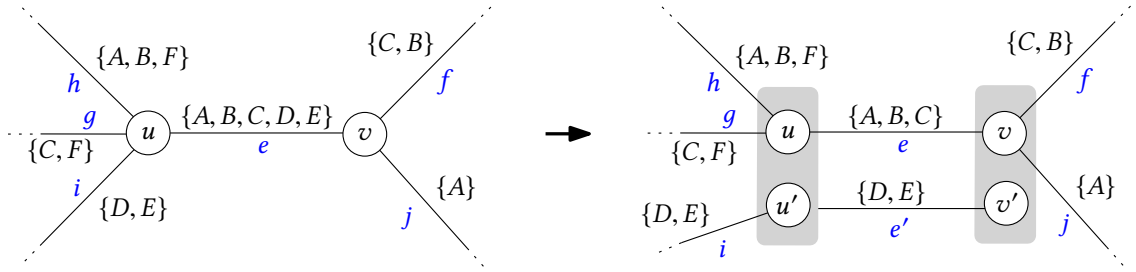


Figure 4.34: Illustration of Untangling Rule 6.

Lemma 4.25. *Untangling Rule 5 does not affect the optimality of the line ordering.*

Proof. The correctness proof is analogous to that of Untangling Rule 5. □

Stump Structures

We now consider situations like depicted in Figure 4.34. It is easy to see that if all lines $L(i)$ on a minor leg i uniquely extend over u into a main leg $e = \{u, v\}$, if i and e are next to each other in the circular edge ordering at u, and if all $L(i)$ have a terminus at v, then we may always place all $L(i)$ on the "outside" of e without any crossings or separations between a line $l_1 \in L(i)$, and a line $l_2 \notin L(i)$. We say i forms an *outer stump* in e.

The following untangling rule captures this intuition:

Untangling Rule 6 (Outer Stump)**.** *Given an edge $e = \{u, v\}$ with $\deg(u) > 2$ and $\deg(v) \geq 2$ and an edge $i = \{w, u\}$. Additionally, the following properties hold:*

1. *Each $l \in L(i)$ uniquely extends over u into e.*
2. *Each $l \in L(i)$ terminates at v.*
3. *$L(i) \neq L(e)$*
4. *Some line $m \in L(e) \setminus L(i)$ extends over v.*
5. *Some line $m \in L(e) \setminus L(i)$ extends over u.*
6. *Either $\pi_e^v(i) = 1$ or $\pi_i^v(e) = 1$.*

Detach the stump by splitting u and v like depicted in Figure 4.34.

Lemma 4.26. *Untangling Rule 6 does not affect the optimality of the line ordering.*

Proof. As all $L(i)$ have a terminus at v, we may place them either at the left or right side of the ordering without introducing any crossings or separations in v. If $\pi_e^v(f) = 1$, placing them on the right side w.r.t. u will induce no crossings or separations between a line $l_1 \in L(i)$ and a line $l_2 \notin L(i)$ at u, and if $\pi_f^v(e) = 1$, placing them on the left side w.r.t. u will induce no such crossings or separations at u. The partial ordering induced by Untangling Rule 6 will thus always be optimal. □

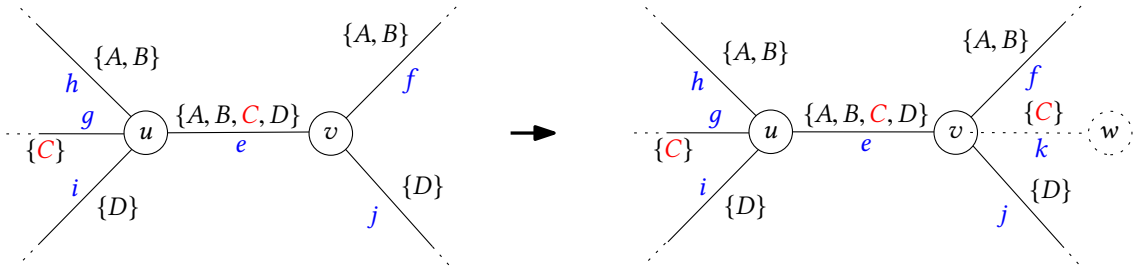


Figure 4.35: Illustration of Untangling Rule 7.

A different kind of stump which we call *inner* stump is depicted in Figure 4.35 and may be transformed into a situation eligible for Untangling Rule 4 or 5 as follows:

Untangling Rule 7 (Inner Stump)**.** *Given an edge $e = \{u, v\}$ which would fulfill every requirement for a full or partial double Y, but there are minor left legs S_l whose lines end at v (but not at u), and/or minor right legs S_r whose lines end at u (but not at v), and which are also not eligible for Untangling Rule 6 (they are* inner *stumps). Let $A : \text{adj}(u) \setminus \{e\} \setminus S_r \mapsto \text{adj}(v) \setminus \{e\} \setminus S_l$ be a bijective mapping which assigns each left minor leg f a right minor leg with $L(f) = L(A(f))$. If the circular edge orderings w.r.t e of the left non-stump minor legs and their corresponding right non-stump minor legs are exactly inverse, add imaginary extensions for u or v for each stump leg as depicted in Figure 4.35, and proceed as in Untangling Rule 4.*

Edge e is now eligible for Untangling Rule 4.

Lemma 4.27. *Untangling Rule 7 does not affect the optimality of the line ordering.*

Proof. As the circular edge orderings w.r.t. e of the non-stump minor legs are exactly inverse, they induce a matching partial ordering without any crossings or separations between two lines l_1 and l_2 from different minor non-stump legs. If we extend the left and right minor legs as described, the induced partial orderings still match on both sides. The partial ordering induced by the subsequent application Untangling Rule 4 is thus optimal. □

A very simple case not yet covered by any of the previous is the following:

Untangling Rule 8 (Double Stump)**.** *Given an edge $e = \{u, v\}$ which is not eligible for Cutting Rule 2 (Terminus Detachment)) and on which one or more lines $l \in L(e)$ have a terminus at u and v. Delete l from $L(e)$ and settle its final position to be at the outside of e.*

Lemma 4.28. *Untangling Rule 7 does not affect the optimality of the line ordering.*

Proof. This follows from the fact that l cannot induce any crossing at u or v, and if placed at the outside of e, can also not induce any separation. □

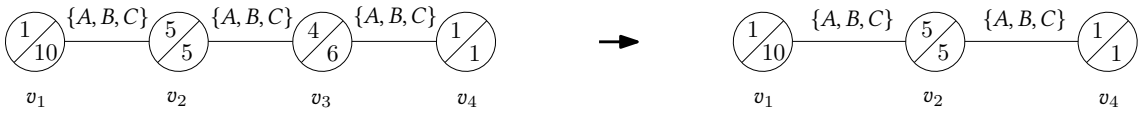


Figure 4.36: A single round of Pruning Rule 1 (Node Contraction) is applied on a line graph G with nodes $v_1, \ldots .v_4$ in the order given by the node indices (from left to right). The crossing and separation penalties between lines A and B in each node are depicted inside the node. During the first round, only v_3 is contracted, but v_2 is now eligible for contraction.

4.9.5 Full Simplification

Naturally, our interest is to combine the pruning, cutting, and untangling rules described so far and apply them as often as possible. This gives rise to several questions, in particular: (1) How often can a simplification rule be applied? (2) Which simplification rules may create new situations eligible for other simplification rules? (3) Is there an adverse order of applying the rules?

This section aims to answer these questions. In particular, we determine upper bounds on the number of times a simplification rule may be applied, and proof several lemmas on their application order. The insight gained is then used to give an algorithm which performs a full simplification (that is, after the algorithm is applied, none of the simplification rules can be applied anymore) with a bounded number of iterations.

We call the standalone application of a simplification rule on all original nodes (Pruning Rule 1, Cutting Rule 2), all original lines (Pruning Rule 2), or all original edges (Pruning Rule 3, Cutting Rule 1, and Untangling Rules 1- 8) a *round* of that rule.

In the general MLNCM-WS case, a single round of Pruning Rule 1 (Node Contraction), applied by simply iterating over each node $v \in |V|$, might result in some remaining nodes eligible for contraction. This happens for simple paths on which all edges have the same lines, and on which all inner nodes have degree 2, as depicted in Figure 4.36, and depends on the ordering in which the nodes are contracted. Observe, however, that such a situation might only occur locally between two nodes that were separated by a previously contracted degree-2 node. For Pruning Rule 1 (Node Contraction), we therefore apply a full round like this: apply Pruning Rule 1 (Node Contraction) to each node $|V|$. If a node v is contracted and was previously adjacent to a node u with $\deg(u) = 2$, immediately apply Pruning Rule 1 (Node Contraction) to u (there are at most 2 such nodes, as $\deg(v) = 2$). We then still process only $O(|V|)$ nodes, but now no opportunity for contraction remains after a single round of Pruning Rule 1 (Node Contraction).

Lemma 4.29. *The order in which we process the lines for Pruning Rule 2 (Line Partner Collapse), the edges for Pruning Rule 3 (Single Edge Prune), or the nodes for Cutting Rules 1 (Single Line Cut) and 2 (Terminus Detachment) in a single standalone round of the respective rule is irrelevant.*

Proof. (1) If Pruning Rule 2 (Line Partner Collapse) would collapse lines $\mathcal{B}$ into a single new line k, then any line set C that would have been eligible for a collapse with any subset of $\mathcal{B}$ is also eligible for a collapse with $\{k\}$. (2) Each edge e eligible for Pruning Rule 3 (Single Edge Prune) is its own graph component, its deletion does not affect other edges. (3) Cutting Rule 1 (Single Line Cut) may neither destroy a new situation eligible for Cutting Rule 1 (Single Line Cut), nor create one (the cut edges had at least one adjacent node with degree 1). (4) Cutting Rule 2 (Terminus Detachment) applied on a node v may neither add, nor remove an opportunity for Cutting Rule 2 (Terminus Detachment) at a node $w \neq v$. □

The following lemma proves that we might also never "destroy" an opportunity for pruning and cutting by an adverse order of application of different types of cutting or pruning rules.

Lemma 4.30. *Given a line graph G and either some node v eligible for Cutting Rule 2 (Terminus Detachment) or Pruning Rule 1 (Node Contraction), or some edge e eligible for Pruning Rule 3 (Single Edge Prune) or Cutting Rule 1 (Single Line Cut). It holds that no application of another pruning or cutting rule might result in a line graph G' in which v (or e) are not eligible anymore for the respective rule.*

Proof. None of the cutting rules requires a minimum number of lines on an edge, so Pruning Rule 2 (Line Partner Collapse) will not destroy any cutting opportunity. Cutting Rule 1 (Single Line Cut) never removes any lines, never adds any lines, never detaches a line from some node v, and never attaches an edge to a node with degree 2, so it will never remove an opportunity for Pruning Rule 3 (Single Edge Prune) or Cutting Rule 2 (Terminus Detachment). Cutting Rule 2 only detaches an edge e not eligible for Cutting Rule 1 (Single Line Cut) from a node v not eligible for Cutting Rule 1 (Single Line Cut), both by definition, and does not affect any other node or edge. Pruning Rule 2 (Line Partner Collapse) only removes nodes not eligible for any other pruning or cutting rule by definition, and leaves line adjacencies at other nodes intact. Pruning Rule 3 (Single Edge Prune) only operates on graph components not eligible for any other pruning or cutting rule per definition. □

Lemma 4.31. *When applied in rounds, Pruning Rule 2 (Line Partner Collapse), Pruning Rule 3 (Single Edge Prune) and Cutting Rules 1-2 are idempotent.*

Proof. (1) Consider a line graph G after a round of Pruning Rule 2 (Line Partner Collapse). If there is a line k which is the result of a line partner collapse and which is now eligible for Pruning Rule 2 (Line Partner Collapse) with a line l, then l would've already been eligible for Pruning Rule 2 (Line Partner Collapse) with any line combined in k, or even with k itself. (2) Pruning Rule 3 (Single Edge Prune) does not create new terminus

nodes. (3) An edge resulting from a cut by Cutting Rules 1 (Single Line Cut) or 2 (Terminus Detachment) is never eligible for Cutting Rules 1 (Single Line Cut) or 2 (Terminus Detachment), as one of its adjacent nodes has now degree 1. □

Lemma 4.32. *Given a line graph G with no situation where Pruning Rule 2 (Line Partner Collapse) may be applied. No cutting, pruning, or untangling rule may result in a line graph G' containing a situation where Pruning Rule 2 (Line Partner Collapse) may be applied.*

Proof. There are two cases to consider. (1) There are no two lines l_1 and l_2 which follow the same path through G, and a cutting, pruning, or untangling rule would create such two lines. (2) Two lines l_1 and l_2 follow the same path p, but p is not simple, and a cutting, pruning, or untangling rule would make p simple. Regarding (1), no cutting, pruning, or untangling rule adds a line to an edge or collapses two edges into one. Additionally, no cutting, pruning, or untangling rule detaches an edge e from a node v over which a line $l \in L(e)$ extends, so no such rule could detach an edge in such a way that a line l_1 would now follow the same path as a line l_2 (Cutting Rule 2 (Terminus Detachment) does only cut an edge, not detach). For the same reason, no such rule could transform a non-simple path p of two lines l_1 and l_2 into a simple path. □

Lemma 4.33. *Given a line graph G with no situation where Cutting Rule 1 (Single Line Cut) may be applied. A single round of Cutting Rule 2 (Terminus Detachment) may never result in a line graph G' containing a situation where Cutting Rule 1 (Single Line Cut) may be applied.*

Proof. Cutting Rule 2 (Terminus Detachment) never decreases the number of lines. □

The same is true in the other direction:

Lemma 4.34. *Given a line graph G with no situation where Cutting Rule 2 (Terminus Detachment) may be applied. A single round of Cutting Rule 1 (Single Line Cut) may never result in a line graph G' containing a situation where Cutting Rule 2 (Terminus Detachment) may be applied.*

Proof. Cutting Rule 1 (Single Line Cut) only introduces terminus nodes at the newly inserted cut nodes, but these have degree 1. □

Lemma 4.35. *Given a line graph G with no situation where a cutting rule may be applied. A single round of Pruning Rule 1 (Node Contraction) may never result in a line graph G' containing a situation where a cutting rule may be applied.*

Proof. Pruning Rule 1 (Node Contraction) never decreases the number of lines per edge, so Cutting Rule 1 (Single Line Cut) may not suddenly be applicable. It also does not introduce terminus nodes, so Cutting Rule 2 (Terminus Detachment) may also not suddenly be applicable. □

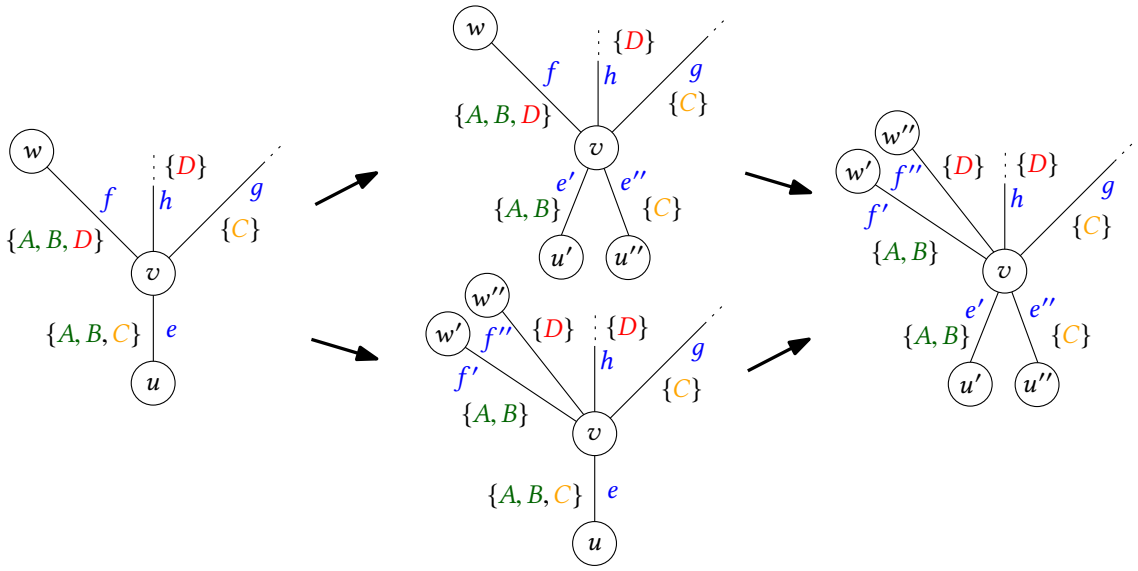


Figure 4.37: Illustration of the proof for Lemma 4.38.

Lemma 4.36. *Given a line graph G with no situation where any cutting or pruning rule, except Pruning Rule 3 (Single Edge Prune), may be applied. A single round of Pruning Rule 3 (Single Edge Prune) may never result in a line graph G' where* any *pruning or cutting rule may be applied.*

Proof. This follows from the fact that each pruned edge is in its own component. □

Corollary 4.37. *Algorithm 4.1 exhaustively prunes and cuts a line graph G.*

Next, we investigate whether there is an adverse application order of the untangling rules, or whether the untangling rules might destroy opportunities for cutting or pruning (or vice versa).

Lemma 4.38. *The order in which a single standalone round of any Untangling Rule is performed (that is, the ordering of the edges or nodes we perform each rule on) is irrelevant.*

Proof. (1) An application of Untangling Rule 1 (Full X) on some node v does not affect any other node, so we only have to consider v itself. Assume we detached an edge pair e, f, and there is an edge g who would've also been eligible for Untangling Rule 1 together with e (or f). But then the lines on $L(e)$ branched at v into f and g, and e and f would not have been eligible for Untangling Rule 1. (2) All other untangling rules always split up a major leg edge e, but leave the minor legs intact. For a full Y, a full double Y, a minor leg edge f could not have been a major leg edge with e as a minor leg edge by definition. (If there was another edge g which, as a major leg, would've been eligible for some untangling rule with f as a minor leg, then f, or an edge f' adjacent to g and with

$G \longleftarrow PruningRule[2](G)$;
$G \longleftarrow CuttingRule[1](G)$;
$G \longleftarrow CuttingRule[2](G)$;
$G \longleftarrow PruningRule[1](G)$;
$G \longleftarrow PruningRule[3](G)$;

Algorithm 4.1: Full pruning and cutting of a line graph G.

the same lines, are still present after applying the rule on e). For a partial Y, a partial double Y, or stump structures, if two adjacent edges e and f with $L(e) \neq L(f)$ share a set of lines $\mathcal{A} = L(e) \cap L(f)$, e might be the minor leg for f, and f might be a minor leg for e (see Figure 4.37). But in such a case, the untangling opportunity on e is still present after f is untangled, and the application order does therefore not matter. □

Similar as for the pruning and cutting rules, the following lemma shows that we might never destroy an untangling opportunity by an adverse order of application of different types of untangling rules.

Lemma 4.39. *Given a line graph G and either some node v eligible for Untangling Rule 1 (Full X), or some edge e eligible for Untangling Rules 2 - 7. It holds that no application of another untangling rule might result in a line graph G' in which v (or e) are not eligible anymore for the respective rule.*

Proof. For Untangling Rules 2 - 7, the argument is the same as (2) in the proof for Lemma 4.38. Untangling Rule 1, may also never destroy an opportunity for Untangling Rules 2 - 7, as two edges e and f eligible for Untangling Rules 1 can never be a minor or major leg for Untangling Rules 2 - 7. The same argument holds for the other direction, and Untangling Rules 2 - 7 may also never destroy an opportunity for Untangling Rule 1. □

Most importantly, we might also never destroy an untangling opportunity by a pruning or cutting rule:

Lemma 4.40. *Given a line graph G and either some node v eligible for Untangling Rule 1 (Full X), or some edge e eligible for Untangling Rules 2 - 7. It holds that no application of any pruning or cutting rule might result in a line graph G' in which v (or e) are not eligible anymore for the respective rule.*

Proof. Let two edges e and f adjacent at v eligible for Untangling Rule 1. By definition, v cannot be eligible for Pruning Rule 1 (Node Contraction), e and f will still be eligible after Pruning Rule 2 (Line Partner Collapse) has collapsed lines on them, Pruning Rule 3 (Single Edge Prune) cannot be applied, Cutting Rule 1 (Single Line Cut) may create novel

edges e' and f' still eligible for Untangling Rule 1 at v, and Cutting Rule 2 by definition cannot destroy the eligibility for Untangling Rule 1. Each remaining untangling rule requires a major leg e with $|L(e)| > 1$, whose lines branch at one (or two) adjacent nodes v (or v, u) into minor legs, rendering the major leg e ineligible for any cutting rule, and v, u ineligible for any pruning rule involving major and/or minor legs. □

Last, we might also never destroy an opportunity for pruning or cutting by an untangling rule:

Lemma 4.41. *Given a line graph G and either some node v eligible for Cutting Rule 2 (Terminus Detachment) or Pruning Rule 1 (Node Contraction), some edge e eligible for Pruning Rule 3 (Single Edge Prune) or Cutting Rule 1 (Single Line Cut), or some set of lines $\mathcal{B}$ eligible for Pruning Rule 2 (Line Partner Collapse). It holds that no application of an untangling rule might result in a line graph G' in which v, e or $\mathcal{B}$ are not eligible anymore for the respective rule.*

Proof. Regarding Cutting Rule 1 (Single Line Cut), no untangling rule adds lines or introduces terminus nodes. Regarding Cutting Rule 2 (Terminus Detachment), no untangling rule adds lines or detaches a terminus edge. Regarding Pruning Rule 1 (Node Contraction), untangling rules also never increase the degree of degree-2 nodes. Regarding Pruning Rule 3 (Single Edge Prune), no untangling rule is eligible for such a single edge structure. Regarding Cutting Rule 2 (Line Partner Collapse), two line sets $\mathcal{A}$ and $\mathcal{B}$ eligible for a line partner collapse can never be split by an untangling rule, as they never branch (by definition). □

For the untangling rules, we also have to consider the fact that the application of a single round of any untangling rule, and the subsequent application of Algorithm 4.1, might create a new situation again eligible for an untangling rule. An example of such a situation is given in Figure 4.38. It might also create new situations eligible for cutting and pruning rules. We therefore have to keep applying all simplification rules in the order given in Algorithm 4.2 until the graph does indeed not change anymore. By the lemmas proved above, we can then be sure that all simplification opportunities have been used, and that we couldn't have arrived at a "simpler" graph with another application order. The following corollary captures this:

Corollary 4.42. *Algorithm 4.2 exhaustively prunes, cuts, and untangles a line graph G.*

Next, we determine an upper bound of the number of outer loop iterations of Algorithm 4.2.

Lemma 4.43. *Given an initially fully pruned and cut line graph G to which a single outer loop iteration of Algorithm 4.2 has been applied. If in the next iteration Untangling Rule 1 (Full X) is applicable, then one (or both) minor leg edges resulted from a major leg split by one of Untangling Rules 2-7 in the previous iteration.*

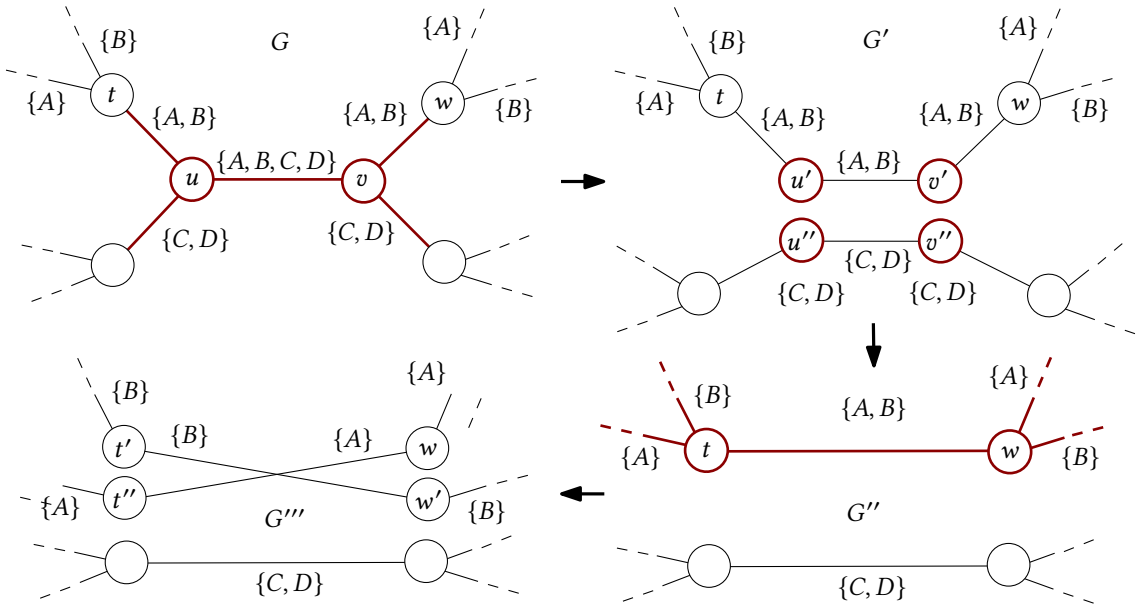


Figure 4.38: Untangling opportunities created by an untangling rule.

Proof. All initial opportunities for Untangling Rule 1 have been resolved in the first iteration of Algorithm 4.2, and no application of Untangling Rule 1 or any pruning or cutting rule may create a novel opportunity for Untangling Rule 1. The novel opportunity was therefore triggered by one of untangling rules 2 - 7. Assume that a minor leg edge of Untangling Rules 2-7 triggered the opportunity. But Untangling Rules 2-7 never change any minor leg edges, and therefore the opportunity must have been created by a split of a major leg edge. □

Lemma 4.44. *Given an initially fully pruned and cut line graph G to which a single outer loop iteration of Algorithm 4.2 has been applied. If in the next iteration there is a new situation eligible for untangling rules 2 - 7, then the line bundle of the major leg of that rule was part of a major leg in the previous iteration.*

Proof. We prove this by contradiction and assume it is not case. Then there are three possibilities: (1) the novel opportunity was created by a combination of pruning and cutting rules, without use of any untangling rule. (2) the major leg resulted from two minor legs of Untangling Rule 1, without use of any other untangling rule. (3) the major leg was already present, and a new *minor* leg enabling the application of the untangling rule was created by untangling rules 2 - 7 in the last iteration.

Regarding (1), we assumed the graph to be initially fully pruned and cut. A standalone application of the pruning and cutting rules would thus not have altered the

$G \longleftarrow PruningRule[2](G)$;
do
  $G^* \longleftarrow G$;
  $G \longleftarrow UntanglingRule[2-6](G)$;
  $G \longleftarrow UntanglingRule[1](G)$;
  $G \longleftarrow CuttingRule[1](G)$;
  $G \longleftarrow CuttingRule[2](G)$;
  $G \longleftarrow PruningRule[1](G)$;
  $G \longleftarrow PruningRule[3](G)$;
while $G^* \neq G$;

Algorithm 4.2: Full pruning, cutting and untangling of a line graph G, with explicit check if the graph is fully simplified.

input graph. Regarding (2), Lemma 4.43 states that the minor leg then resulted from a major leg of untangling rules 2 - 7. Regarding (3), if a *minor* leg edge f for untangling rules 2 - 7 resulted from the split of a major leg edge in the previous iteration, this would either mean that a minor leg edge with the same lines (and therefore the untangling opportunity) was already present in the previous iteration, or that f was already eligible for untangling in the previous iteration (see the example in Figure 4.37). □

Lemma 4.45. *Given an initially fully pruned and cut line graph G. Then the number of iterations of Algorithm 4.2 is bounded by $M-1$, where M is the maximum number of lines per edge in G.*

Proof. This directly follows from Lemmas 4.43 and 4.44. At each iteration, the minor leg(s) of Untangling Rule 1 or the major legs of Untangling Rule 2-7 resulted from a split of a major leg edge in the previous iteration. For each application of an untangling rule, there is thus a path of previous major leg splits back to the initial input state of Algorithm 4.2, and this path has a maximum length of $M-1$ (a set of M lines can only be split $M-1$ times). □

This is reflected in Algorithm 4.3 (because we assumed the graph to be initially pruned and cut, we use M iterations).

Corollary 4.46. *Algorithm 4.3 exhaustively simplifies a line graph G.*

It remains to show that a full simplification may be performed in reasonable time. The next section will show that it is possible to fully simplify any given input line graph G in time polynomial only in the maximum number M of lines per edge, and the maximum node degree D.

$i \longleftarrow M$;
$G \longleftarrow PruningRule[2](G)$;
while $i > 0$ **do**
  $G \longleftarrow UntanglingRule[2-6](G)$;
  $G \longleftarrow UntanglingRule[1](G)$;
  $G \longleftarrow CuttingRule[1](G)$;
  $G \longleftarrow CuttingRule[2](G)$;
  $G \longleftarrow PruningRule[1](G)$;
  $G \longleftarrow PruningRule[3](G)$;
  $i \longleftarrow i - 1$;
end

Algorithm 4.3: Full pruning, cutting and untangling of a line graph G, with a guaranteed number of outer loop iterations.

4.9.6 Complexity

We first examine the complexity of applying a single round of each pruning, cutting or untangling rule on some graph G. Afterwards, we analyze the complexity of Algorithm 4.3.

By $D = \max_{v \in V} \deg(v)$ we again denote the maximum node degree of our input line graph G. By $M = \max_{e \in E} |L(e)|$ we again denote the maximum number of lines per segment. We may remove nodes with degree 0 from the line graph prior to optimization and thus assume that $2|E|$ is an upper bound for $|V|$.

For our analysis, we additionally make the following assumptions:

1. Each line is identified by a numerical ID.
2. Each edge is identified by a numerical ID.
3. Each node is identified by a numerical ID.
4. V is ordered by the node IDs (as is typically the case if nodes are stored in a continuous array).
5. For each e, $L(e)$ is ordered by the line IDs
6. For each v, adj(v) is ordered by the outgoing angle of the adjacent edges.
7. For a line $l \in \mathcal{L}$ and an edge e, we can check whether $l \in L(e)$ in constant time (for example, by building a lookup array).
8. For a line $l \in \mathcal{L}$ and a node v, we can check whether v is a terminus for l in constant time (for example, by storing this information per node).
9. We can retrieve the lines $\mathcal{T}_v$ that terminate at a node v in constant time (for example, by storing this information beforehand).
10. For an edge $e = \{u, v\}$, we can check in constant time whether for all $l_1, l_2 \in L(e)^2, l_1 \neq l_2, w_\times(l_1, l_2, u) \leq w_\times(l_1, l_2, v)$ or vice versa.

11. For an edge $e = \{u, v\}$, we can check in constant time whether for all $l_1, l_2 \in L(e)^2, l_1 \neq l_2, w_{||}(l_1, l_2, u) \leq w_{||}(l_1, l_2, v)$.

The last two points need some additional explanation. We can allow constant time lookups of these properties if we initially determine and store tuples $(w^{\min}_{\times u}, w^{\max}_{\times u})$ and $(w^{\min}_{||u}, w^{\max}_{||u})$ for both sides of all edges $e = \{u, v\} \in E$, where $w^{\min}_{\times u}$ is the minimum cost $w_\times(l_1, l_2, u)$ over all $l_1, l_2 \in L(e)^2, l_1 \neq l_2$, $w^{\max}_{\times u}$ is the maximum such cost, and so on. These values can be determined in $O(|E|M^2)$ time. Then to determine if for example $w_\times(l_1, l_2, u) \leq w_\times(l_1, l_2, v)$ for all $l_1, l_2 \in L(e)^2, l_1 \neq l_2$, it suffices to check whether $w^{\max}_{\times u} \leq w^{\min}_{\times v}$. When two edges are merged during the application of Pruning Rule 1 (Node Contraction), we update the tuples accordingly, which can be done in constant time. Note that when an edge and an adjacent node v are split in one of the Untangling Rules, an update is not required, as we must always consider the original input node v^* for the crossing and separation weights to ensure optimality, as explained in Section 4.9.2.

Complexity of Pruning Rules

For a single application of Pruning Rule 1 (Node Contraction) to some node v with $\deg(v) = 2$, we have to check whether adjacent edges e and f for a node v with $\deg(v) = 2$ have $L(e) = L(f)$, and whether the crossing and separation weights at the remaining adjacent nodes are equal or smaller. This can be done in $O(M)$, as we assumed that (1) both $L(e)$ and $L(f)$ are sorted and (2) the check whether the crossing and separation weights are smaller or equal can be done in constant time. In a single round of Pruning Rule 1 (Node Contraction) we have to contract $O(|V|)$ nodes, as described in Section 4.9.5. A single round of Pruning Rule 1 (Node Contraction) can therefore be applied in $O(|E|M)$.

For Pruning Rule 2 (Line Partner Collapse), we use the following algorithm:

(1) Iterate over each $v \in V$ (recall that V is ordered by the node IDs) (2) For each $l \in \mathcal{T}_v$, mark l as processed and follow its path through G until the path either terminates, diverges, or arrives at v again. Remember the visited node IDs as well as their crossing and separation weights. If the path terminates, append the list of visited nodes (with their crossing and separation weights) to a list A and continue. If the path was not simple, drop it, and continue. (3) Sort A and extract the line clusters with equal path and weights. (4) For each such line cluster $\mathcal{B}$ with $|\mathcal{B}| > 1$, choose any $l \in \mathcal{B}$ as a reference line, update the crossing weights for l and delete all $\mathcal{B} \setminus \{l\}$ from the line graph.

For steps (1) and (2) we have to follow at most $|E|$ edges at most M times, so these can be done in $O(|E|M)$. A can be sorted in $O(|E|M \log M)$, and matching line clusters can then be retrieved in $O(|E|M)$. Finally, the line collapsing in step (4) can be done in $O(|E|M)$, so we need time $O(|E|M \log M)$ overall.

For Pruning Rule 3 (Single Edge Prune), we iterate over all edges, deleting edges where both adjacent nodes have a degree of 2. This can be done in $O(|E|)$.

Complexity of Cutting Rules

Cutting Rule 1 (Single Line Cut) has to check for each edge e whether $|L(e)| = 1$ and then cut this edge, adding two new nodes and edges. This can be done in $O(|E|)$.

As we assumed that we can check whether a line ends at a node in constant time, we can retrieve all candidates for Cutting Rule 2 (Terminus Detachment) in $O(|E|)$. The edge detachment can then be done in constant time.

Complexity of Untangling Rules

To apply Untangling Rule 1 (Full X) to a single node, a naive way would be to test for all pairs $\{e, f\}$ of adjacent edges whether $L(e) = L(f)$. As there are at most $\binom{D}{2} < D^2$ such pairs, this can be done in $O(MD^2)$, resulting in a time complexity of $O(|V|MD^2) = O(|E|MD^2)$ for a full round of Untangling Rule 1 (Full X).

However, as we assumed $L(e)$ to be sorted for each e, we may sort $L(e_1), \ldots, L(e_{\deg(v)})$ for adjacent edges $e_1, \ldots, e_{\deg(v)}$ in $O(MD \log D)$ and extract matching clusters of size 2 in $O(MD)$, improving the complexity for a full round of Untangling Rule 1 (Full X) to $O(|V|MD \log D) = O(|E|MD \log D)$.

For applying Untangling Rule 2 (Full Y) to a single node, we have to identify a major leg e (this has a time complexity of $O(D)$) and check whether the lines of the remaining adjacent edges (the minor legs) are contained in $L(e)$. This takes $O(MD)$. To check whether the minor legs $f_1, \ldots, f_N$ are pairwise disjoint, we first do a k-way merge of the $L(f_i)$ and then check whether $|\bigcup L(f_i)| = \sum |L(f_i)|$. This takes $O(MD \log D)$. Applying Untangling Rule 2 (Full Y) to all $v \in V$ thus takes $O(|V|MD \log D) = O(|E|MD \log D)$.

Assume we apply Untangling Rule 3 (Partial Y) to a single node v, given a major leg candidate e. We have to collect all adjacent edges $f_i \neq e$ which share a line with e, and for each such f_i check which lines are shared. This can again be done in $O(MD)$ via a simple list intersection. Checking whether these shared line sets are pairwise disjoint can then again be done in $O(MD \log D)$ like for Untangling Rule 2 (Full Y). Applying Untangling Rule 3 (Partial Y) to all $v \in V$ therefore also takes $O(|E|MD \log D)$.

Double-Y structures may be understood as two compatible Single-Y structures, connected by the bijective mapping m described in Untangling Rule 4 (Full Double Y). Applying Untangling Rule 4 (Full Double Y) to an edge $e = \{u, v\}$ therefore amounts to applying the check for Untangling Rule 2 (Full Y) for u and v (assuming that e as been "cut") and finding the bijective mapping m. The former can be done in $O(MD \log D)$ for a single edge e. As we assumed both $\text{adj}(u)$ and $\text{adj}(v)$ to be sorted, the mapping m can be found in $O(MD)$ time and $O(M)$ space by creating two arrays A_u and A_v of size M. For each minor leg edge f_i adjacent to u, we then take a random line l from $L(f_i)$ (as the minor leg edges are pairwise disjoint, this is a unique identifier), write the clockwise position of f_i to $A_u[l]$. We do the same at v. Afterwards, we iterate over A_u, check the

corresponding entry in A_v for line equivalency and write the position pair to our mapping m. The complexity of applying Untangling Rule 4 (Full Double Y) to all $v \in V$ is therefore $O(|E|MD \log D)$.

For partial Double-Y structures, the argumentation is similar. W.l.o.g., applying Untangling Rule 3 (Partial Y) to an edge $e = \{u, v\}$ amounts to applying the check for Untangling Rule 2 (Full Y) for u, applying the check for Untangling Rule 3 (Partial Y) to v and finding the bijective mapping m. The complexity of applying Untangling Rule 3 (Partial Y) to all $v \in V$ is therefore also $O(|E|MD \log D)$.

To collect all outer stumps for Untangling Rule 6 (Outer Stump) on a single edge e, we have to check for four adjacent edges f_1, f_2, f_3, f_4 whether they are minor legs for an outer stump in the worst case. This can be done in $O(M)$. A single round of Untangling Rule 6 (Outer Stump) therefore has a complexity of $O(|E|M)$.

Checking whether Untangling Rule 7 (Inner Stump) is applicable to a major leg edge e has the same worst-time complexity as checking whether e is applicable for Untangling Rule 4 (Full Double Y) or Untangling Rule 3 (Partial Y). The (imaginary) edge extension can be done in constant time. Untangling Rule 7 (Inner Stump) therefore has the same time complexity as Untangling Rules 7 (Inner Stump) and 4 (Full Double Y), which is $O(|E|MD \log D)$.

Untangling Rule 8 (Double Stump) can be applied to a single edge e in time $O(M)$ as we assumed that we can check in constant time whether a line terminates at a node v. Untangling Rule 8 (Double Stump) therefore has a complexity of $O(|E|M)$.

Table 4.2 gives an overview of the complexities of each simplification rule.

Upper Bounds on Intermediate Graph Sizes

Algorithm 4.3 iteratively applies all simplification rules but Pruning Rule 2 (Line Partner Collapse) to an *intermediate* graph G^*. Let $\hat{M}$ be the maximum number of lines, and $\hat{D}$ be the maximum node degree of these graphs (including the original graph G). Similarly, let $\hat{N}$ be the maximum number of edges of any graph.

As we never merge edges with different lines, $\hat{M} = M$. The untangling rules might split an edge and increase both the number of edges and the degree of an adjacent node, but this may only happen M times per edge. In the worst case, the cutting rules might double the number of edges. We therefore have $\hat{D} \leq DM$ and $\hat{N} \leq |E|2M$.

Complexity of Full Simplification

Under the initial assumptions made above, the loop of Algorithm 4.3 therefore has a worst-case asymptotic complexity of $O(M\hat{N}\hat{M}\hat{D} \log \hat{D}) = O(|E|M^4 D \log MD)$. As we have $O(|E|M \log M) \subset O(|E|M^4 D \log MD)$, the single application of Pruning Rule 2 (Line

Table 4.2: Asymptotic complexities of single rounds of pruning, cutting, and untangling rules

Rule	Label	Complexity
Pruning Rule 1	Node Contraction	$\mathcal{O}(\|E\|M)$
Pruning Rule 2	Line Partner Collapse	$\mathcal{O}(\|E\|M \log M)$
Pruning Rule 3	Single Edge Prune	$\mathcal{O}(\|E\|)$
Cutting Rule 1	Single Line Cut	$O(\|E\|)$
Cutting Rule 2	Terminus Detachment	$O(\|E\|)$
Untangling Rule 1	Full X	$\mathcal{O}(\|E\|MD \log D)$
Untangling Rule 2	Full Y	$\mathcal{O}(\|E\|MD \log D)$
Untangling Rule 3	Partial Y	$\mathcal{O}(\|E\|MD \log D)$
Untangling Rule 4	Full Double Y	$\mathcal{O}(\|E\|MD \log D)$
Untangling Rule 5	Partial Double Y	$\mathcal{O}(\|E\|MD \log D)$
Untangling Rule 6	Outer Stump	$\mathcal{O}(\|E\|M)$
Untangling Rule 7	Inner Stump	$\mathcal{O}(\|E\|MD \log D)$
Untangling Rule 8	Double Stump	$\mathcal{O}(\|E\|M)$

Partner Collapse) does not change the asymptotic complexity of Algorithm 4.3, which therefore remains $\mathcal{O}(|E|M^4 D \log MD)$.

For all real-world datasets, both M and D are usually very small (in our biggest dataset, M was 9, and D was 4) and may be assumed constants. Under this additional assumption, Algorithm 4.3 runs in linear time.

4.10 Full Solve Through Simplification

For some line graph instances, applying the full simplification described above already yields an optimal line ordering. This section describes several (non-trivial) classes of line graph instances for which this is the case.

4.10.1 Tree-Like Line Graphs

We first define the following type of line graph, for which Figure 4.39 gives an example.

Definition 4.7 (Tree-Like Line Graph)**.** A line graph $G = (V, E, L, \mathcal{L})$ is called tree-like if the following holds: (1) G is a tree. (2) Each $l \in \mathcal{L}$ follows a simple path through G. (3) Each $l \in \mathcal{L}$ terminates at the root node. (4) Each $l \in \mathcal{L}$ terminates at a leaf node.

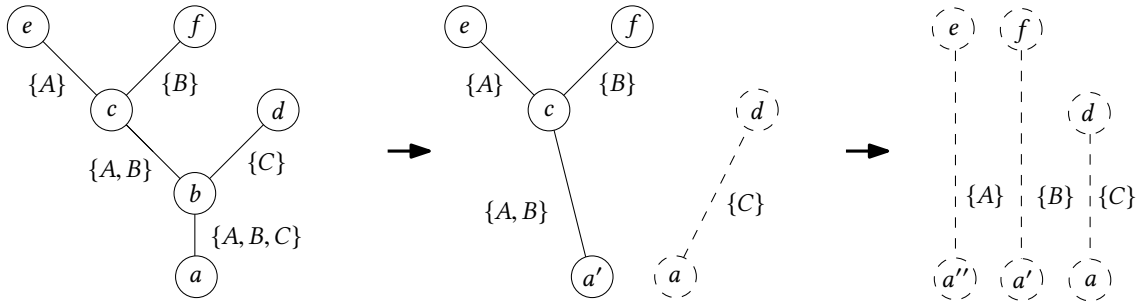


Figure 4.39: A tree-like line graph G is simplified into a forest of single-edge trees by Algorithm 4.3, which are pruned by Pruning Rule 3 (Single Edge Prune), but displayed here with dashed lines for better understanding. Algorithm 4.3 therefore already finds an optimal line ordering for such instances.

Lemma 4.47. *Given a tree-like line graph G with uniform crossing and separation weights. Algorithm 4.3 yields an empty line graph G' (and thus an optimal line ordering for G).*

Proof. The initial pruning will remove any two lines l_1, l_2 with the same path through G, as well as any inner nodes of degree 2. We can thus ignore these cases and focus on trees with inner nodes of degree greater than 2, with no two lines following the same path. Then two cases remain: (1) The root node r has degree 1. (2) The root node r has a degree greater than 1. In the second case, as all lines terminate at the root, Cutting Rule 2 (Terminus Detachment) will detach all edges from r, leaving a forest of $\deg(r)$ trees with roots of degree 1. We therefore only have to consider case (1). Given then such a tree-like line graph G with root r and $\deg(r) = 1$, Untangling Rule 2 (Full Y) may immediately be applied to the only child node v of r, leaving a forest of $\deg(v)$ tree-like line graphs. This process continues until the tree has been untangled into a forest of single-edge graphs, pruned by Pruning Rule 3 (Single Edge Prune). Figure 4.39 gives an example. □

Corollary 4.48. *Given a tree-like line graph G with uniform crossing and separation weights. An optimal MLNCM-WS solution for G can be found in $O(|E|M^4D \log MD)$.*

Corollary 4.49. *Given a tree-like line graph G. An optimal MLNCM-S solution for G can be found in $O(|E|M^4D \log MD)$.*

Corollary 4.50. *Given a tree-like line graph G uniform crossing weights. An optimal MLNCM-W solution for G can be found in $O(|E|M^4D \log MD)$.*

Corollary 4.51. *Given a tree-like line graph G. An optimal MLNCM solution for G can be found in $O(|E|M^4D \log MD)$.*

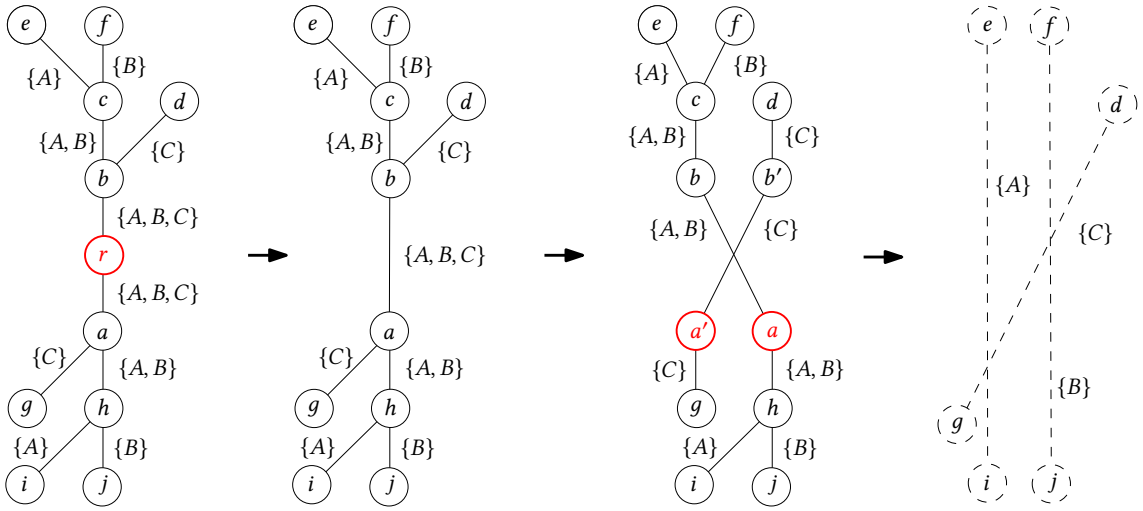


Figure 4.40: Simplifying a two-sided tree-like line graph (with root r) by first applying Pruning Rule 1 (Node Contraction), and then Untangling Rule 2 (Full Y). This results in a forest of two-sided tree-like line graphs, which can be similarly simplified.

4.10.2 Two-Sided Tree-Like Line Graphs

Definition 4.8 (Two-Sided Tree-Like Line Graphs)**.** A line graph $G = (V, E, L, \mathcal{L})$ is called two-sided tree-like (2-tree-like) if the following holds: (1) G is a tree. (2) The root of G has degree 2, the tree under the left root child is the left subtree, the tree under the right root child is the right subtree. (3) Each $l \in \mathcal{L}$ follows a simple path through G. (4) Each $l \in \mathcal{L}$ continues through the root node. (5) Each $l \in \mathcal{L}$ has a terminus at a leaf node of the left subtree, and a terminus at a leaf node of the right subtree. (6) The left and right subtree are branch-symmetric: for each branch of two or more line bundles at level n in the left subtree there is a corresponding node in the right subtree where the same line bundles branch.

Lemma 4.52. *Given a 2-tree-like line graph G with uniform crossing and separation weights. Applying Algorithm 4.3 to G will yield an empty line graph G' (and thus an optimal line ordering for G).*

Proof. The root node r will be initially contracted by Pruning Rule 1 (Node Contraction), leaving an edge e to which Untangling Rule 2 (Full Y) can be applied. This leaves a forest of double-sided tree-like line graphs, which are either linear with every edge sharing the same set of lines, or again a non-trivial instance of a two-sided tree-like line graph. In the former case, the line graph will be pruned away, in the latter case, the process recursively continues (Figure 4.40). □

Corollary 4.53. *Given a 2-tree-like line graph G with uniform crossing and separation weights. An optimal MLNCM-WS solution for G can be found in $O(|E|M^4 D \log MD)$.*

Corollary 4.54. *Given a 2-tree-like line graph G. An optimal MLNCM-S solution for G can be found in $O(|E|M^4D \log MD)$.*

Corollary 4.55. *Given a 2-tree-like line graph G uniform crossing weights. An optimal MLNCM-W solution for G can be found in $O(|E|M^4D \log MD)$.*

Corollary 4.56. *Given a 2-tree-like line graph G. An optimal MLNCM solution for G can be found in $O(|E|M^4D \log MD)$.*

4.10.3 Real-World Examples

Two-sided tree-like line graphs are not only of theoretical interest, but are for example often found in S-Bahn systems. These systems typically feature a central main corridor which bundles all lines, which then branch into the suburbs on both sides of the corridor. Figure 4.41, shows the official 2021 network map of the S-Bahn system in Stuttgart, for which Algorithm 4.3 finds an optimal line-ordering solution with 2 crossings and 0 separations if the separation and crossing weights are uniform.

4.11 Experimental Evaluation

We have implemented all the methods described above: the full exhaustive search (Exh), the greedy approach described in Section 4.6.3 (Gr), the greedy approach with lookahead described in Section 4.6.3 (GrLa), GrLa with subsequent hillclimbing (GrLa+Hill), GrLa with subsequent simulated annealing (GrLa+Ann), both simulated annealing and hill climbing on a random initial line ordering (Hill, Ann), and the baseline ILP formulation (b-ILP), and the improved ILP formulation (i-ILP). We tested them on 7 network datasets of increasing complexity (see Table 4.3 for details): the 2016 tram network of Freiburg, the 2016 light-rail network of Dallas, the 2016 light-rail network of Chicago, the 2015 (before heavy construction started) light-rail network of Stuttgart, the 2020 light-rail network of Sydney, the 2016 tram network of Turin, and the 2016 subway network of New York. Note that the Stuttgart network evaluated in this section is the light rail network, *not* the (much simpler) S-Bahn network given in Figure 4.41. The datasets were generated from raw GTFS data by our approaches from Chapters 2 and 3. Artifacts left during this process were manually fixed to ensure a fair comparison. For the New York subway network, we increased the network complexity by explicitly considering express lines (not done in the official network map). Evaluations were run on an Intel Xeon X5560 machine with 8 cores (each with 2.8 GHz) and 36 GB of RAM.

For each dataset and method, we evaluated the weighted setting without punishing line separations (MLNCM-W) and the weighted setting with punishing line separations (MLNCM-WS). If line separations were considered, the methods are given in a starred

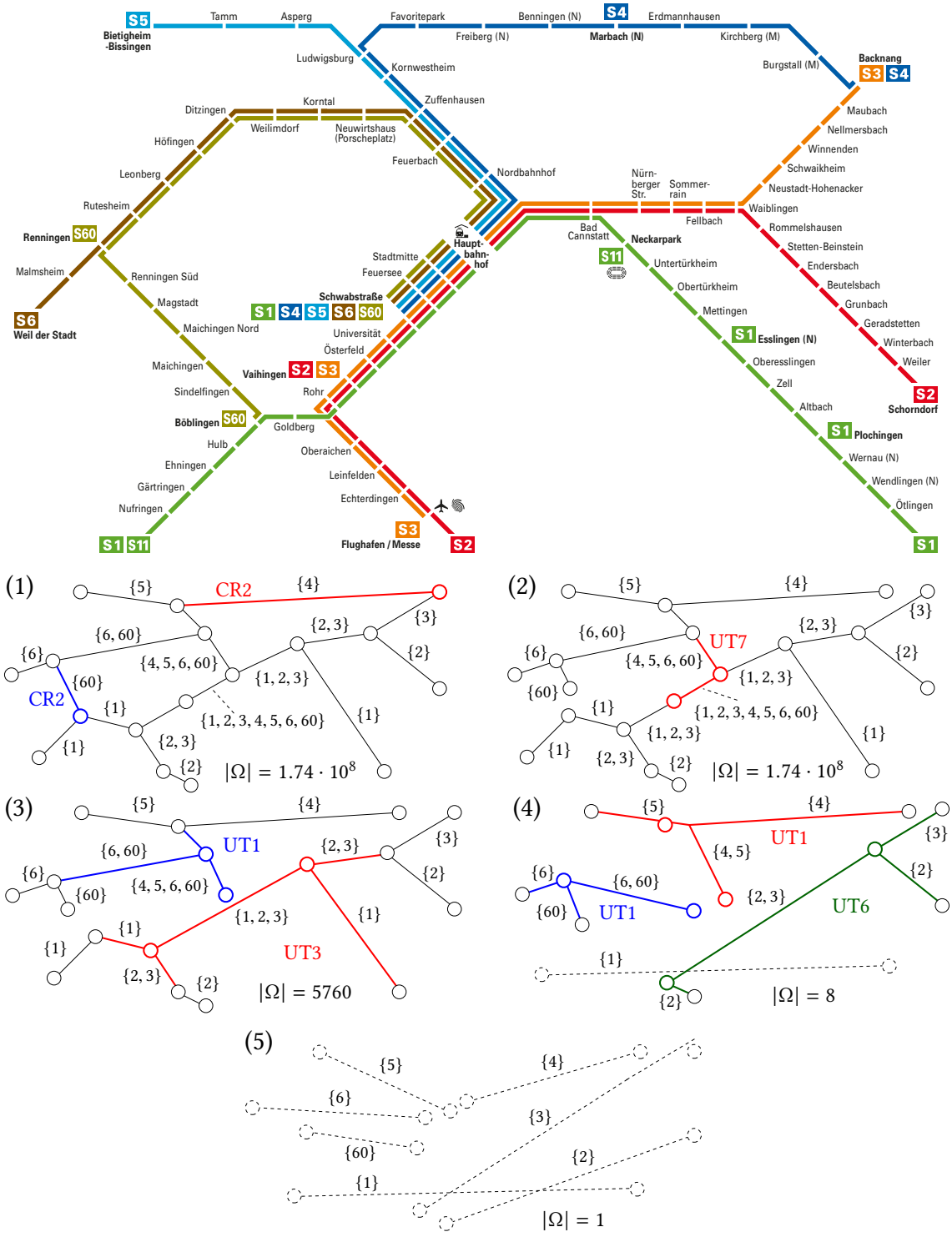


Figure 4.41: Top: Official 2021 S-Bahn system map of Stuttgart. For uniform crossing and separation weights, a full simplification by Algorithm 4.3 gives an optimal line-ordering solution with 0 separations and 2 crossings. Bottom: Intermediate line graphs after several iterations of Algorithm 4.3 (Lines S1 and S11 were considered a single line S1, and nodes contracted by Pruning Rule 1 (Node Contraction) are not depicted for brevity). The applied rules (CR = Cutting Rule, UT = Untangling Rule) are highlighted.

Table 4.3: Line graphs used in our experimental evaluation. Under $|\mathcal{S}|$ we give the number of station nodes in the graph. Under $|V|$ we give the numer of nodes (including station nodes). Under $|E|$ we give the number of edges. Under $|\mathcal{L}|$ we give the number of lines. M is the maximum number of lines per edge. D is the maximum node degree. $|\Omega|$ is the line-ordering search space size on the original line graph.

	$\lvert\mathcal{S}\rvert$	$\lvert V\rvert$	$\lvert E\rvert$	$\lvert\mathcal{L}\rvert$	M	D	$\lvert\Omega\rvert$
Freiburg (FR)	74	76	79	5	4	4	3×10^{9}
Dallas (DA)	108	117	118	7	4	4	6×10^{20}
Chicago (CG)	143	153	154	8	6	4	4×10^{33}
Sydney (SD)	175	198	202	9	6	3	9×10^{58}
Sydney (ST)	192	218	228	15	8	4	1×10^{105}
Turin (TU)	339	398	435	14	5	4	2×10^{89}
New York (NY)	456	517	548	26	9	5	2×10^{269}

version (for example, i-ILP optimizes MLNCM-W using our improved ILP formulation, and i-ILP* optimizes MLNCM-WS).

To ensure a realistic evaluation setup, we used manually adjusted crossing and separations weights which proved to produce line orderings that are very close to those in maps by professional designers, as will be shown in Section 4.11.5. The weights were as follows: At a node v, split-crossings at non-station nodes were weighted by $1 \times \deg(v)$, split-crossings at station nodes were weighted by $3 \times \deg(v)$, intra-path crossings at non-station nodes were weighted by $4 \times \deg(v)$, intra-path crossings at station nodes where weighted by $12 \times \deg(v)$, separations at non-station nodes were weighted by $3 \times \deg(v)$, separations at station nodes were weighted by $9 \times \deg(v)$. Additionally, we set a special weight W higher than *any* of the weights above for crossings and separations at degree-2 stations.

Section 4.11.1 gives an overview of our main results. In Section 4.11.2 we compare the results of our heuristic approaches described in Section 4.6. The performance of our ILP on three different solvers (GLPK, COIN-OR CBC, and gurobi) will be evaluated in Section 4.11.3. Section 4.11.4 will then evaluate the effect of the simplification rules described in Section 4.9 on both the heuristic approaches, and the linear programs. The full evaluation setup can be found online[1].

4.11.1 Main Results

Our main results are summarized in Tables 4.4 and 4.5. The key takeaways are as follows:

[1] `https://github.com/ad-freiburg/loom-eval`

Table 4.4: Main results for the running times of a selection of our heuristic and exact (based on an ILP) methods on all test datasets. Under 'graph' we give the simplification applied to the input graph: either none (raw), pruning & cutting (pr/cut) or full simplification (f-simp).

	heuristic					exact				
method graph	GrLa* raw	Hill* raw	Hill* pr/cut	Hill* f-simp	Ann* f-simp	b-ILP* raw	b-ILP* f-simp	i-ILP* raw	i-ILP* pr/cut	i-ILP* f-simp
FR	**1ms**	4ms	2ms	0.8ms	21ms	1.2s	4ms	21ms	18ms	**4ms**
DA	**1ms**	14ms	4ms	0.8ms	58ms	0.9s	54ms	31ms	15ms	**6ms**
CG	**2ms**	50ms	11ms	9ms	0.4s	2h	32.1s	0.7s	**0.2s**	0.3s
SD	**3ms**	0.1s	31ms	29ms	0.5s	—	34m	1.1s	0.6s	**0.6s**
ST	**5ms**	0.5s	0.1s	21ms	0.6s	—	48.4s	5.8s	1.2s	**0.4s**
TU	**14ms**	0.4s	0.1s	74ms	2.7s	30m	1m	1.7s	0.7s	**0.5s**
NY	**25ms**	3.4s	0.7s	86ms	1.0s	—	8m	20.4s	2.1s	**0.6s**

Heuristic approaches are not good enough. While some heuristic approaches (in particular GrLa) give good results in terms of quality in the MLNCM-W setting, the results were generally unsatisfactory for MLNCM-WS.

The improved ILP is much faster. Our improved ILP formulation (i-ILP) is several orders of magnitude faster than the baseline formulation (b-ILP), in both the MLNCM-W and MLNCM-WS setting. For many datasets, the baseline ILP could not be optimized in under 6 hours.

Line graph simplification helps a lot. Full line graph simplification using the pruning, cutting, and untangling rules described above greatly reduced the optimization time. In particular, when compared to only applying the relative simple pruning and cutting rules, the untangling rules provided an additional speed up. Line graph simplification also had a positive effect on the quality of our approximation approaches.

Perfect optimization is possible in an interactive setting. With full line graph simplification applied, we were able to solve all test datasets to optimality in under 600 milliseconds, which enables our method to be used in an interactive setting (for example a map editor).

For more speed, combine greedy approach with hillclimbing. If an ILP is not an option, the greedy lookahead method with hillclimbing is a viable alternative.

Table 4.5: Main results for the relative approximation error $\eta = \frac{\theta_{\text{approx}}}{\theta} - 1$ of a selection of our heuristic methods on all test datasets.

method	Gr*	GrLa*	GrLa*	GrLa+ Hill*	GrLa+ Hill*	GrLa+ Ann*	GrLa+ Ann*	Hill*	Hill*	Ann*	Ann*
graph	raw	raw	f-simp	raw	f-simp	raw	f-simp	raw	f-simp	raw	f-simp
FR	0.7	0.9	0.9	0.9	0.5	2.3	0.1	6.1	0.2	1.7	0.1
DA	1.7	1.0	1.0	1.0	1.0	23.3	0.3	44.2	2.2	25.5	0.7
CG	1.2	0.3	0.4	0.3	0.4	5.1	0.1	7.3	0.5	3.6	0.1
SD	2.3	0.9	1.7	0.9	1.3	6.0	1.7	10.8	3.2	6.1	1.2
ST	2.2	0.1	0.1	0.1	0.1	7.3	0.3	11.5	1.1	7.0	0.3
TU	3.2	1.0	1.7	0.7	0.4	4.0	0.8	7.0	1.4	3.8	0.6
NY	10.1	1.9	2.0	0.9	0.5	23.2	0.7	38.4	1.8	21.4	0.8
avg	3.0	0.9	0.9	0.7	0.6	10.2	0.6	17.9	1.5	9.9	**0.6**

4.11.2 Comparison of Heuristic Approaches

We evaluated the heuristic approaches described in Section 4.6 on all datasets and measured the running time, the final objective value θ, and the relative approximation error η when compared to the optimal solution obtained via our ILP. The relative approximation error η is defined as follows:

$$\eta = \frac{\theta_{\text{approx}}}{\theta} - 1, \tag{4.23}$$

where θ is the optimal objective value (a relative approximation error of 1 means that the solution is twice as bad as the optimal solution). For each input line graph, we evaluated both the full unmodified baseline graph, and the pruned and cut line graph (the effect of additionally applying the untangling rules will be evaluated in Section 4.11.4). For probabilistic approaches (Ann, GrLa+Ann) and hill climbing with random initialization the results are the averages of 10 optimization runs. The results are given in Table 4.6. For brevity, we only give the results for 3 datasets there. The average results over all datasets are given in Table 4.7.

Solution Times

On the raw input graph, an exhaustive search (Exh) could not find a solution in under 6 hours even on our smallest dataset (Freiburg). Using a carefully optimized implementation of the target function calculation method described in Section 4.6.1, we were able to test around 30,000 line ordering solutions per second on average, but even with that

Table 4.6: Comparison of our approximate approaches on the raw input graph, and the pruned & cut graph. We give the solution space size Ω, the time t, the number of crossings $\times$, the number of separations $||$, the target function value θ, and the relative approximation error η.

		On raw input graph						On pruned & cut graph					
		$\lvert\Omega\rvert$	t	$\times$	$\lvert\rvert$	θ	η	$\lvert\Omega\rvert$	t	$\times$	$\lvert\rvert$	θ	η
FR	Exh		—	—	—	—	—		0.1 s	3.0	2.0	21.0	0.0
	Gr		0.4 ms	5.0	1.0	39.0	0.9		0.1 ms	5.0	1.0	39.0	0.9
	GrLa		0.4 ms	3.0	2.0	21.0	0.0		0.1 ms	3.0	2.0	21.0	0.0
	+Hill	3×10^{9}	1 ms	3.0	2.0	21.0	0.0	5×10^{3}	0.5 ms	3.0	2.0	21.0	0.0
	+Ann		0.2 s	5.8	1.4	61.2	1.9		52 ms	4.8	1.6	42.0	1.0
	Hill		5 ms	10.3	2.9	159.0	6.6		1 ms	6.6	1.3	55.5	1.6
	Ann		0.2 s	6.0	2.1	63.9	2.0		52 ms	3.6	1.4	29.1	0.4
	Exh*		—	—	—	—	—		0.1 s	5.0	0.0	39.0	0.0
	Gr*		0.5 ms	5.0	1.0	66.0	0.7		0.1 ms	5.0	1.0	66.0	0.7
	GrLa*		0.4 ms	3.0	2.0	75.0	0.9		0.1 ms	3.0	2.0	75.0	0.9
	+Hill*	3×10^{9}	1.0 ms	3.0	2.0	75.0	0.9	5×10^{3}	0.6 ms	3.0	2.0	75.0	0.9
	+Ann*		0.2 s	7.3	1.5	130.2	2.3		42 ms	4.8	1.4	81.9	1.1
	Hill*		4 ms	11.1	2.4	278.1	6.1		2 ms	6.2	0.9	78.0	1.0
	Ann*		0.1 s	7.0	1.3	104.7	1.7		41 ms	5.0	1.2	75.3	0.9
SD	Exh		—	—	—	—	—		—	—	—	—	—
	Gr		3 ms	24.0	15.0	183.0	1.0		0.5 ms	27.0	13.0	319.0	2.5
	GrLa		3 ms	12.0	1.0	183.0	1.0		0.6 ms	17.0	5.0	225.0	1.5
	+Hill	9×10^{58}	6 ms	12.0	1.0	183.0	1.0	8×10^{20}	7 ms	12.0	3.0	111.0	0.2
	+Ann		2.7 s	34.2	24.7	435.5	3.8		2.0 s	18.7	10.4	124.7	0.4
	Hill		0.1 s	52.5	38.7	728.7	7.1		44 ms	29.4	13.3	208.5	1.3
	Ann		2.6 s	38.7	25.6	448.4	4.0		1.8 s	19.0	11.5	127.7	0.4
	Exh*		—	—	—	—	—		—	—	—	—	—
	Gr*		3 ms	24.0	15.0	354.0	2.3		0.5 ms	27.0	13.0	526.0	3.9
	GrLa*		3 ms	12.0	1.0	210.0	0.9		0.6 ms	17.0	5.0	288.0	1.7
	+Hill*	9×10^{58}	6 ms	12.0	1.0	210.0	0.9	8×10^{20}	6 ms	18.0	5.0	249.0	1.3
	+Ann*		1.3 s	47.4	21.6	754.5	6.0		0.6 s	26.7	7.8	261.2	1.4
	Hill*		0.1 s	60.8	29.4	1278.6	10.8		31 ms	34.6	11.0	406.3	2.8
	Ann*		1.3 s	42.6	22.5	762.6	6.1		0.6 s	27.0	9.2	264.1	1.4
NY	Exh		—	—	—	—	—		—	—	—	—	—
	Gr		22 ms	76.0	32.0	1107.0	6.9		3 ms	70.0	14.0	342.0	1.4
	GrLa		25 ms	51.0	13.0	294.0	1.1		3 ms	50.0	15.0	309.0	1.2
	+Hill	2×10^{269}	99 ms	47.0	12.0	234.0	0.7	1×10^{92}	35 ms	45.0	12.0	171.0	0.2
	+Ann		11.7 s	156.6	102.5	2431.0	16.2		7.1 s	76.0	32.0	354.9	1.5
	Hill		3.7 s	214.6	144.6	4566.7	31.4		0.8 s	104.1	53.0	603.9	3.3
	Ann		12.1 s	158.9	103.4	2526.6	16.9		7.4 s	74.3	36.5	360.6	1.6
	Exh*		—	—	—	—	—		—	—	—	—	—
	Gr*		22 ms	76.0	32.0	2133.0	10.1		3 ms	70.0	14.0	612.0	2.2
	GrLa*		25 ms	51.0	13.0	564.0	1.9		3 ms	50.0	15.0	597.0	2.1
	+Hill*	2×10^{269}	0.1 s	44.0	10.0	357.0	0.9	1×10^{92}	56 ms	45.0	13.0	351.0	0.8
	+Ann*		6.8 s	178.4	80.6	4655.1	23.2		3.5 s	96.4	24.5	736.8	2.8
	Hill*		3.4 s	251.8	108.0	7557.9	38.4		0.7 s	143.1	39.9	1344.9	6.0
	Ann*		8.0 s	181.2	77.3	4299.9	21.4		3.7 s	96.1	25.1	722.4	2.8

Table 4.7: Average solution time (t) and relative approximation errors (η) of all our heuristic approaches, over all datasets.

	On baseline graph		On pruned & cut graph	
	t	η	t	η
Gr	**7ms**	2.3	**0.9ms**	1.3
GrLa	7ms	0.4	1.0ms	0.6
+Hill	25ms	**0.4**	14ms	**0.1**
+Ann	4.5s	6.9	2.8s	0.8
Hill	0.7s	13.1	0.2s	1.6
Ann	4.3s	6.5	2.8s	0.7
Gr*	**7ms**	3.0	**0.8ms**	1.8
GrLa*	7ms	0.9	1.0ms	1.1
+Hill*	23ms	**0.7**	13ms	**0.7**
+Ann*	2.6s	10.2	1.5s	1.5
Hill*	0.6s	17.9	0.1s	2.5
Ann*	2.6s	9.9	1.5s	1.4

throughput, it would take more than a day to explore the entire search space. On the pruned & cut graph, however, we were able to find optimal solutions using an exhaustive search for Freiburg and Dallas in under 100 ms.

The greedy approaches (Gr, GrLa) were very fast, with an average solution time of 7 ms. On a randomized initial ordering solution, Hill converged to local optimum in under 0.6 second on average with line separations considered, which improved to 0.1 seconds on the pruned and cut graph. Simulated annealing (Ann) was slower, with an average solution time of 2.6 seconds on the raw, and 1.5 seconds on the pruned & cut graph, both with line separations considered. We note that we did only little tweaking of the annealing parameters. Faster cooling might lead to drastically improved solution times for Ann.

If we combined the greedy lookahead (GrLa) approach with hill climbing and simulated annealing, GrLa+Hill converged in under 30 ms on average for all settings (unfortunately, this also meant that only little improvement was made, see the next section for details). The solution times for GrLa+Ann remained roughly the same.

Quality

For the Freiburg dataset, the greedy approach with lookahead (GrLa) produced an optimal line ordering if no line separations were considered. For all other datasets, the

target function values were typically much worse than the optimal values in the raw input graph. For example, on the raw graph of the New York subway dataset with line separations considered, even the best-performing heuristic approach (GrLa+Hill) had a relative approximation error of 0.9, meaning that the target function was nearly 2 times higher than the optimal one. If GrLa was combined with hill climbing, the quality improved only a little. If GrLa was combined with simulated annealing on the raw graph, the quality greatly deteriorated. It seems likely that the simulated annealing quickly leaves the solution provided by the initial GrLa approach because of the initial high temperature and cannot find its way back. If the graph was pruned and cut first, this effect was extenuated, but still measurable.

In general, pruning and cutting greatly improved the quality of the local search approaches using a random initial ordering, but had only little effect on the greedy approaches. This was to be expected, though: neither the pruned degree-2 nodes, nor the cut edges with only one line play any role in the decision process for finding the relative ordering between two lines in the GrLa approach.

On average, the greedy lookahead method combined with subsequent hillclimbing (GrLa+Hill) performed best, achieving an average relative approximation error of only 0.2 in the MLNCM-W settings. If line separations were considered, however, the target function values were, on average, nearly twice as high.

4.11.3 Comparison of Linear Programs

We evaluated all our testing datasets on three different ILP solvers: the GNU Linear Programming Kit 4.65 (GLPK), COIN-OR CBC 2.10.3, and gurobi 9.1.2 (GLPK and COIN-OR CBC are open source and publicly available, gurobi requires a license). For all solvers, we used the default parameters (GLPK, gurobi) or reconstructed the default command line parameters for the library (COIN-OR CBC). The evaluation results for our ILP variants (b-ILP, b-ILP*, i-ILP, i-ILP*) on the raw input graph as well as on the pruned & cut graph can be seen in Table 4.8. As for the heuristic approaches, the effects of the untangling rules on the solution times will be evaluated separately in Section 4.11.4.

We observed a clear (and expected) performance ranking of the ILP solvers: gurobi usually required the lowest solution times, closely followed by COIN-OR. GLPK followed with some distance. This performance gap only emerged in larger ILPs, though, as the various pre-solving techniques applied by COIN-OR and gurobi often only seem to pay off for larger ILPs. In general, our baseline ILP formulation proved to be impractical on the raw input graph. We were not able to optimize most of the datasets with any solver in under 6 hours, except Freiburg and Dallas. Pruning & cutting did not help a lot here, and even if it did, solution times on the best performing solver (gurobi) were still up to several minutes.

In contrast, we were already able to optimize every dataset using the raw input graph

Table 4.8: ILP Dimensions (given as rows × cols) and solution times for all our ILP variants on the raw input graph and on the pruned & cut input graph. If a graph had multiple components, we optimized them separately, and the dimensions for the largest component are given, but solution times are always cumulative. We only measured the time to solve the ILP, not the ILP construction time or the time it took to extract the line ordering from the ILP solution (which were both negligible) or the time required to set up the solver environment.

		On raw graph				On pruned graph					
		rows×cols	GLPK	CBC	GU	rows×cols	GLPK	CBC	GU	×	||
FR	b-ILP	3.0k×268	13.5s	11.4s	0.4s	1.1k×96	0.4s	3.4s	0.1s	3	2
	b-ILP*	5.7k×324	52.4s	46.8s	1.2s	2.0k×114	1.0s	8.1s	0.2s	5	0
	i-ILP	399×297	0.1s	22ms	8ms	144×105	10ms	13ms	8ms	3	2
	i-ILP*	487×339	0.6s	0.1s	21ms	182×123	25ms	53ms	18ms	5	0
DA	b-ILP	4.8k×500	1m	39.5s	1.0s	1.4k×161	1.4s	8.0s	0.2s	3	1
	b-ILP*	8.9k×638	2m	1m	0.9s	2.6k×195	1.6s	7.5s	0.2s	3	0
	i-ILP	799×572	0.3s	99ms	23ms	252×181	22ms	16ms	9ms	3	0
	i-ILP*	991×662	1.8s	0.2s	31ms	317×211	0.1s	25ms	15ms	3	0
CG	b-ILP	38.2k×861	—	—	48m	8.3k×265	—	1h	26.5s	16	5
	b-ILP*	72.8k×1.1k	—	—	2h	15.4k×311	—	1h	44.7s	21	0
	i-ILP	1.4k×982	33.1s	0.6s	51ms	393×284	0.1s	0.2s	13ms	16	4
	i-ILP*	1.9k×1.2k	2h	19.7s	0.7s	504×337	45.6s	4.1s	0.2s	21	0
SD	b-ILP	97.2k×1.4k	—	—	5h	50.1k×537	—	—	4m	13	5
	b-ILP*	182.7k×1.9k	—	—	—	94.0k×699	—	—	1h	13	0
	i-ILP	2.4k×1.6k	2m	0.7s	99ms	873×598	9.2s	0.4s	45ms	13	5
	i-ILP*	3.2k×2.0k	1h	33.7s	1.1s	1.2k×756	11m	5.7s	0.6s	13	0
ST	b-ILP	206.6k×2.4k	—	—	—	44.2k×928	—	—	2h	39	14
	b-ILP*	379.6k×3.2k	—	—	—	83.1k×1.1k	—	—	—	—	—
	i-ILP	4.2k×2.8k	9m	2.1s	0.1s	1.5k×1.0k	20.7s	0.2s	33ms	39	9
	i-ILP*	5.7k×3.6k	—	7m	5.8s	2.1k×1.3k	—	34.7s	1.2s	40	5
TU	b-ILP	23.3k×2.2k	—	—	8m	11.7k×897	—	—	33.4s	45	7
	b-ILP*	43.0k×2.8k	—	—	30m	21.7k×1.1k	—	—	3m	50	2
	i-ILP	3.5k×2.5k	52m	0.8s	91ms	1.4k×986	2m	0.4s	64ms	45	10
	i-ILP*	4.5k×3.0k	—	27.3s	1.7s	1.8k×1.2k	—	12.2s	0.7s	50	2
NY	b-ILP	248.0k×6.0k	—	—	—	95.8k×2.3k	—	—	—	—	—
	b-ILP*	467.3k×7.9k	—	—	—	179.5k×2.8k	—	—	—	—	—
	i-ILP	10.2k×7.0k	—	3.5s	0.2s	3.7k×2.6k	—	2.3s	100ms	43	17
	i-ILP*	13.6k×8.6k	—	14m	20.4s	4.9k×3.2k	—	1m	2.1s	48	3

Table 4.9: Effects of full line graph simplification on line graph dimensions. $|V|$ is the number of nodes, $|E|$ is the number of edges, M is the maximum number of lines per edge, $|\Omega|$ is the search space size (sum of the search space sizes of the graph components), C is the number of nontrivial (more than 2 nodes) graph components, C^1 is the number of nontrivial graph components with a search space size of 1 (not requiring further optimization).

	Raw input graph						Pruned & cut graph						Fully simplified graph					
	$\|V\|$	$\|E\|$	M	$\|\Omega\|$	C	C^1	$\|V\|$	$\|E\|$	M	$\|\Omega\|$	C	C^1	$\|V\|$	$\|E\|$	M	$\|\Omega\|$	C	C^1
FR	76	79	4	3×10^{9}	1	0	36	30	4	5×10^{3}	4	0	19	15	4	2×10^{2}	1	0
DA	117	118	4	6×10^{20}	2	1	44	35	4	8×10^{6}	4	3	38	27	4	2×10^{3}	4	3
CG	153	154	6	4×10^{33}	1	0	27	26	6	5×10^{9}	2	1	24	22	6	1×10^{9}	2	1
SD	196	202	6	9×10^{58}	1	0	64	56	6	8×10^{20}	5	2	51	42	6	6×10^{19}	3	2
ST	218	228	8	1×10^{105}	1	0	84	79	8	2×10^{37}	6	2	69	58	6	8×10^{12}	2	0
TU	398	435	5	2×10^{89}	1	0	237	202	5	6×10^{36}	19	12	203	165	5	5×10^{31}	16	12
NY	517	548	9	2×10^{269}	2	1	173	176	9	1×10^{92}	9	1	119	96	6	4×10^{36}	3	1
med red										1×10^{38}						1×10^{39}		

with our improved ILP on an open-source solver (CBC), although it still took several minutes on larger instances. However, an important takeaway is that in the MLNCM-WS setting without any prior line graph simplification, even a highly sophisticated solver like gurobi has solution times too long for interactive use (for example, the New York network took over 20 seconds to optimize if line separations were considered).

Pruning & cutting already helped a lot and enabled us to solve every instance of MLNCM-WS to optimality using our improved ILP in under 2.1 seconds (the best free solver (CBC) still required over a minute on the largest dataset, New York).

4.11.4 Effects of Full Line Graph Simplification

We have seen above that the pruning & cutting rules already help a lot in two ways: they sped up the solution times of both the approximate approach and the ILPs (sometimes dramatically), and often also improved the average quality of the approximate approaches. In this section, we evaluate the effect of an additional application of the untangling rules described in Section 4.9.4.

Table 4.9 gives an overview of the effect of full line graph simplification when compared to only applying pruning & cutting. While pruning & cutting already had a tremendous effect on the search space size (reducing it by nearly 200 orders of magni-

Table 4.10: Impact of full simplification on ILP sizes and solution times. We only measured the time to solve the ILP, not the ILP construction time or the time it took to extract the line ordering from the ILP solution (which were both negligible) or the time required to set up the solver environment.

		Pruned graph				Fully simplified graph			
		rows×cols	GLPK	CBC	GU	rows×cols$_{max}$	GLPK	CBC	GU
FR	i-ILP	144 ×105	10ms	13ms	8ms	0 ×0	4ms	4ms	4ms
	i-ILP*	182 ×123	25ms	53ms	18ms	0 ×0	4ms	4ms	4ms
DA	i-ILP	252 ×181	22ms	16ms	9ms	123 ×90	7ms	10ms	7ms
	i-ILP*	317 ×211	0.1s	25ms	15ms	149 ×102	13ms	13ms	6ms
CG	i-ILP	393 ×284	0.1s	0.2s	13ms	371 ×267	0.3s	0.1s	15ms
	i-ILP*	504 ×337	45.6s	4.1s	0.2s	482 ×320	27.9s	4.0s	0.3s
SD	i-ILP	873 ×598	9.2s	0.4s	45ms	837 ×572	5.4s	0.2s	42ms
	i-ILP*	1.2k ×756	11m	5.7s	0.6s	1.2k ×727	25m	10.2s	0.6s
ST	i-ILP	1.5k ×1.0k	20.7s	0.2s	33ms	497 ×349	0.8s	0.2s	24ms
	i-ILP*	2.1k ×1.3k	—	34.7s	1.2s	678 ×435	1m	13.5s	0.4s
TU	i-ILP	1.4k ×986	2m	0.4s	64ms	1.2k ×854	51.4s	0.3s	45ms
	i-ILP*	1.8k ×1.2k	—	12.2s	0.7s	1.6k ×1.0k	—	10.5s	0.5s
NY	i-ILP	3.7k ×2.6k	—	2.3s	100ms	1.4k ×990	11.4s	0.2s	50ms
	i-ILP*	4.9k ×3.2k	—	1m	2.1s	1.9k ×1.2k	—	8.6s	0.6s

tude for the New York dataset), the additional application of the untangling rules again gave a reduction of up to 57 orders of magnitude for the largest dataset, with a median search space size reduction of 39 orders of magnitude. While all input test datasets consisted of only a single nontrivial graph component with a search space size greater than 1, the pruning and cutting rules increased the number of nontrivial components to up to 19 (for Turin). The untangling rules then decreased the number of components again. For example, the Stuttgart light rail network had 6 nontrivial components after pruning & cutting, but only 2 after the additional application of the untangling rules. This is because the full line graph simplification completely untangled some components left by the pruning & cutting rules into trivial components consisting of only a single edge, an effect already demonstrated in the example of the simpler Stuttgart S-Bahn network given in Figure 4.41.

The effect of full line graph simplification on the heuristic approximation approaches

Table 4.11: Impact of full simplification on selected baseline heuristic solution times and objective function values.

		Pruned graph					Fully simplified graph				
		t	$\times$	$\|\|$	θ	η	t	$\times$	η	$\|\|$	θ
FR	Exh*	0.1 s	5.0	0.0	39.0	0.0	4 ms	5.0	0.0	39.0	0.0
	GrLa*	0.1 ms	3.0	2.0	75.0	0.9	< 1 ms	3.0	2.0	75.0	0.9
	+Hill*	0.6 ms	3.0	2.0	75.0	0.9	0.5 ms	3.0	1.0	60.0	0.5
	+Ann*	42 ms	4.8	1.4	81.9	1.1	21 ms	4.8	0.2	43.8	0.1
	Hill*	2 ms	6.2	0.9	78.0	1.0	0.8 ms	4.2	0.4	48.0	0.2
	Ann*	41 ms	5.0	1.2	75.3	0.9	21 ms	5.0	0.1	41.7	0.1
SD	Exh*	—	—	—	—	—	—	—	—	—	—
	GrLa*	0.6 ms	17.0	5.0	288.0	1.7	0.5 ms	17.0	5.0	288.0	1.7
	+Hill*	6 ms	18.0	5.0	249.0	1.3	6 ms	18.0	5.0	249.0	1.3
	+Ann*	0.6 s	26.7	7.8	261.2	1.4	0.4 s	29.1	9.4	290.6	1.7
	Hill*	31 ms	34.6	11.0	406.3	2.8	29 ms	34.6	12.4	451.5	3.2
	Ann*	0.6 s	27.0	9.2	264.1	1.4	0.5 s	23.6	9.7	242.1	1.2
NY	Exh*	—	—	—	—	—	—	—	—	—	—
	GrLa*	3 ms	50.0	15.0	597.0	2.1	1 ms	50.0	13.0	570.0	2.0
	+Hill*	56 ms	45.0	13.0	351.0	0.8	23 ms	49.0	9.0	282.0	0.5
	+Ann*	3.5 s	96.4	24.5	736.8	2.8	1.0 s	59.8	9.5	329.7	0.7
	Hill*	0.7 s	143.1	39.9	1344.9	6.0	86 ms	82.0	18.2	538.0	1.8
	Ann*	3.7 s	96.1	25.1	722.4	2.8	1.0 s	61.2	9.7	342.4	0.8
avg	GrLa*					1.1					1.1
	+Hill*					0.7					0.6
	+Ann*					1.5					0.6
	Hill*					2.5					1.5
	Ann*					1.4					0.6

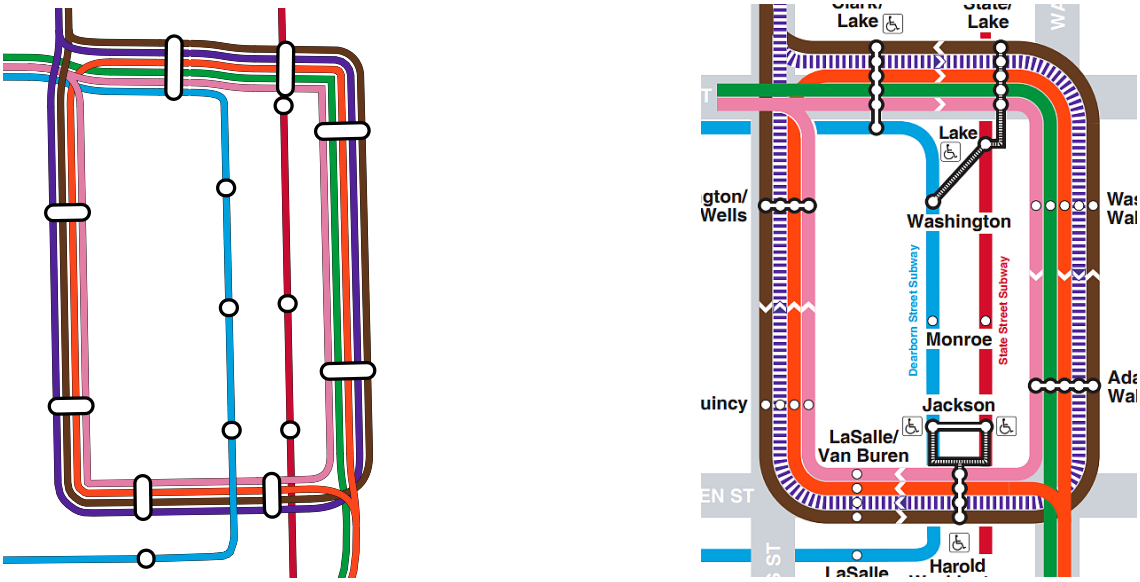


Figure 4.42: Left: Optimal line ordering for the Chicago Loop using our MLNCM-WS formulation. Right: Line ordering in the official map. A single line swap of the blue and brown line between the lower right and the upper left intersection transforms our line ordering into the official line ordering. This was the only line swap required on the entire map.

was twofold: First, we measured an additional speedup across all techniques. Second, full simplification resulted in an additional quality boost for nearly all our methods. With full simplification, the average approximation error for our informed greedy search with lookahead combined with subsequent local search (hill climbing) slightly went down to 0.6 from 0.7 on the pruned & cut input graph. For other approaches, the effect was more pronounced: using simulated annealing on the fully simplified graph gave a relative average approximation error of 0.6, compared to 1.4 on the pruned & cut graph, and 9.9 on the raw input graph.

4.11.5 Comparison to Manually Designed Maps

An evaluation of the esthetic quality of our line orderings is difficult as there are no established quality metrics (except for the number of crossings). Nevertheless, we compared the optimal line orderings of the Freiburg, Dallas, Chicago, and Stuttgart networks to existing official maps, created by professional map designers. We compared them to our MLNCM-WS approach (with line crossings and line separations penalized), using the manually tuned crossing and separation weights mentioned above. As the official maps and their line orderings are not available in machine-readable form, we hand-counted the crossings and separations from published PDF maps and calculated the value of the

	Official map			MLNCM-WS			
	×	||	θ	×	||	θ	T
FR	7	1	132	6	0	48	2
DA	3	1	27	3	0	9	1
CG	26	0	80	27	0	80	1
ST	65	5	264	64	2	156	4

Table 4.12: Comparison of the line orderings in professionally designed (official) maps, and the line orderings produced by our approach. Under T we give the number of line swaps necessary to transform one ordering into the other (we only counted swaps between the same lines along a path once).

target function by hand. For Dallas and Chicago, a single line swap was enough to transform our ordering into the official ordering (see Figure 4.42 for an example). Even the highly complex Stuttgart network only required 4 line swaps.

4.12 Conclusions and Future Work

This chapter discussed the problem of finding optimal line orderings in metro maps, a crucial task for ensuring both readability and esthetic quality of the final map. We gave a formulation of the classic Metro Line Crossing Minimization (MLCM) problem, namely the Metro Line Node Crossing Minimization (MLNCM) problem, which only allows crossings at nodes, not on edges. To produce maps that are more esthetically pleasing, and to give map designers some control over the crossing placements, we introduced 3 variants of MLNCM: a weighted version (MLNCM-W), a version in which line separations are penalized (MLNCM-S), and a version in which both crossings and separations are weighted (MLNCM-WS). We have shown that MLNCM, MLNCM-W, and MLNCM-WS are NP-hard on general graphs, while the complexity of MLNCM-S is on open problem. We gave a linear time algorithm for MLNCM on graphs in which all lines follow simple paths and terminate at nodes of degree 1. Even on these instances, MLNCM-W remains NP-hard. We gave several heuristic approximation algorithms for the most general problem (MLNCM-WS), and gave a baseline ILP to solve the problem to optimality. As the baseline ILP required impractically high solution times, we formulated an improved variant.

To further speed up the optimization, we described several pruning, cutting, and untangling rules which may be applied to the line graph before optimization. Their benefit lies in the fact that they may not only reduce the search space size by many orders of magnitude, but also typically split the line graph into connected components, enabling parallel optimization. We proved that none of these simplification rules affect the optimality of the final line ordering and showed their effectiveness in our evaluation. We described instances for which the simplification rules already find an optimal line ordering, as they decompose the input graph into components with a search space sizes

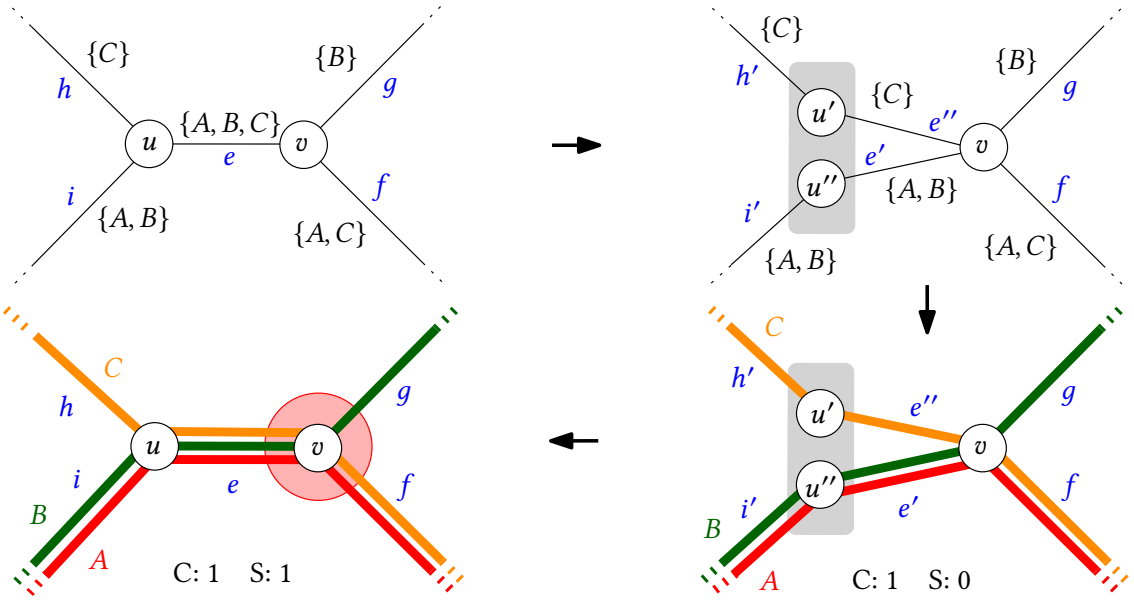


Figure 4.43: A generic ripping rule like described in this section only works if line separations are not considered. In this example, applying the ripping rule to node u may later hide a separation of lines A and C at v, resulting in a non-optimal line ordering.

of 1. Finally, we showed that with carefully chosen crossing and separation weights, the line orderings found by our approach closely match those of professionally designed maps.

We see several directions for future work. First, we do not claim that the line graph simplification rules given in Section 4.9 are exhaustive. During the development of our prototype, several ideas for additional untangling rules emerged: "Ripping" casts the intuition behind our informed greedy search with lookahead from Section 4.6.4 into an explicit untangling rule. Consider Figure 4.43: if the lines on an edge $e = \{u, v\}$ branch at u into n minor legs, split u into n nodes like in Untangling Rule 4 or 5 if all crossings and separations may occur at equal or cheaper cost at v. The only difference to Untangling Rule 5 is that we do not require the minor leg mapping A. In this configuration, however, a key property used in the correctness proofs of the previous untangling rules is lost: there might now be separations between two lines from different minor legs at v, and these separations will be hidden if u is split. Still, such a ripping rule may be applied if line separations are not considered, or if line separation costs at v are 0.

The untangling rules described in Section 4.9.4 were built on the concepts of *minor* and *major* legs. Each leg was a distinct edge. We may generalize this into an abstract variant we call *induced* minor leg. The main difference is that an induced minor leg may consist of multiple edges. Figure 4.44 gives an example for an induced full X: although there is no opportunity for any of the untangling rules defined above, *sets* of edges may

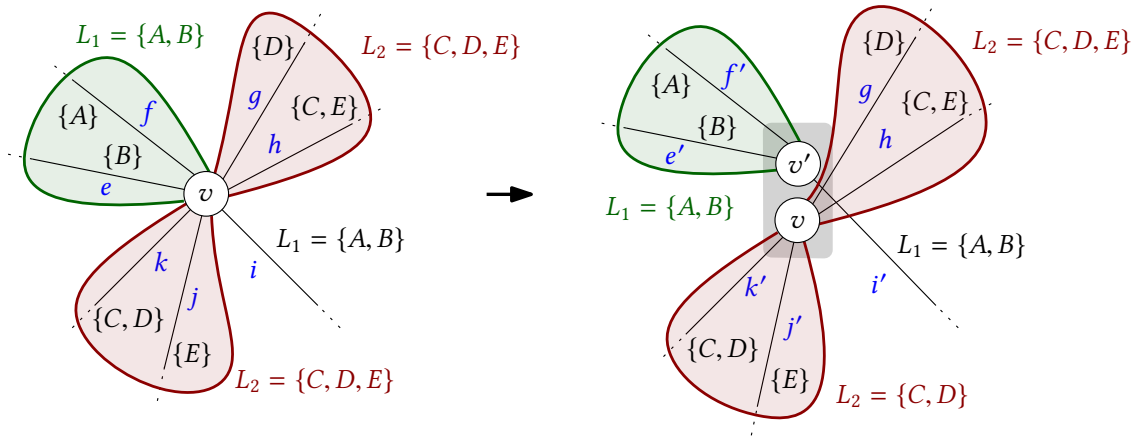


Figure 4.44: Application of an induced untangling rule on a node v. Although no adjacent edge fulfills the requirements for a leg in Untangling Rule 1, the pair of edge sets $\{e, f\}$ and $\{i\}$ does. Generally speaking, induced untangling rules search for a grouping of adjacent edges in such a way that group sets may act as minor or major legs in the untangling rules.

satisfy the requirements for a leg. For instance, edge i and the set of edges $\{e, f\}$ share the exact same lines and may therefore be detached from v' using a similar argument as in the correctness proof for Untangling Rule 1. Such an *induced* version can be given for all our untangling rules and may be able to further break down the line graph by creating novel simplification opportunities.

As mentioned above, the crossing and separation weights used in our evaluation are based on intuition and manual fine-tuning. A user study (for example, A/B testing) could help to conclusively determine the weights which are both most esthetically pleasing and informative.

With full line graph simplification using our pruning, cutting, and untangling rules, we were able to optimize MLNCM-WS instances with realistic crossing and separations weights in under 600 milliseconds for all testing datasets, enabling the use of our methods in an interactive setting. For settings where this is still not fast enough, a combination of an informed greedy search and subsequent hillclimbing on the fully simplified line graph was shown to be a practical and fast alternative of often sufficient quality.

Figure 4.45: Excerpt from a line-ordering optimized rendering of the New York subway network, optimized by our ILP in 0.6s. Line graph generated by our pipeline described so far, rendered using the approach from Chapter 6 by the *transitmap* tool (Chapter 7).

Chapter 5

Schematic Transit Maps

So far, we have only considered geographically accurate transit maps in which lines follow their real-world geographic course as close as possible. While these maps are especially well suited to be laid over existing maps or satellite imagery, the generation of *schematic* transit maps is also of high practical interest. Transit maps which are posted in stations or in vehicles, or are distributed in print usually show a high degree of simplification to ensure legibility. The de facto standard for schematized transit maps follows the classic octilinear layout already mentioned in Chapter 1. The problem of generating such octilinear transit maps automatically has been studied extensively over the last decades. In recent years, a number of maps following alternative layouts have been published both by transit agencies and map enthusiasts [110]. This has led to an interest in the automatic generation of transit maps which are not octilinear, but for example orthoradial [13, 113] or curvilinear [58, 48]. Although this section will begin with a description of an approach to render octilinear maps, we will later extend our method to other layouts. We consider the following problem:

Problem 5 (Line Graph Schematization)**.** *Given a line graph $G = (V, E, L, \mathcal{L})$, find a schematic drawing $\mathcal{D}_G = (P, C)$ of G such that $P(v) \in \mathbb{R}^2$ assigns a position to every node $v \in V$ and $C(e) = (q_0, q_1, \ldots, q_{n-1}), q_i \in \mathbb{R}^2$ assigns a piece-wise linear curve to every edge $e \in E$. $\mathcal{D}_G$ should resemble a classic schematic transit map.*

The following section will make it clear how the schematic transit map resemblance is formalized. After some preliminaries and a discussion of related work, we will first describe and evaluate an approach to generate schematic transit map drawings in a classic octilinear fashion by finding a set of optimal interconnected shortest paths on a specially crafted octilinear base grid graph. We will then continue to extend our approach to other base grids, both to further speed up the original approach and to render maps which are not octilinear. The methods presented in this chapter have been previously published in [21] and [22].

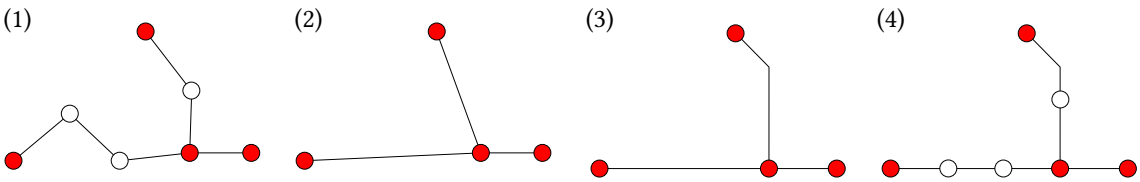


Figure 5.1: The degree-2 heuristic typically applied to transit networks before schematization. Degree-2 nodes in the initial input line graph (1) are contracted (2), the resulting line graph is schematized (3), and contracted nodes are re-inserted equidistantly (4).

5.1 Preliminaries

To formally define a drawing as a schematic transit map, a set of *hard* and *soft* constraints is often used. The soft constraints constitute the objective function, and the hard constraints define the set of feasible solutions. While these constraints have been originally defined for octilinear embeddings [117, 118], they also provide a solid foundation for other schematization types. As hard constraints, we thus consider the following:

- H1 *Schematic Representation.* Each edge curve $C(e)$ must follow the schematization layout defined for the drawing type (e.g. octilinear, orthoradial, ...).
- H2 *Topology Preservation.* The topology of the original input line graph must be preserved. In particular, this means that no crossings between edges must be introduced, non-incident edges must not share common points and circular edge orderings around nodes must be preserved.
- H3 *Map Density.* Distances between stations and curve anchor points must be higher than a threshold D.

As soft constraints, we consider the following:

- S1 *Edge Monotony.* Edge bends should be minimized and obtuse angles should be preferred.
- S2 *Edge Length.* The total length of edges should be minimized.
- S3 *Geographical Accuracy.* The station positions should reflect the original geographical positions.

Our running assumption in this chapter is that these soft and hard constraints are sufficient to model the esthetic quality and readability of a map.

5.1.1 Degree-2 Heuristic

Soft constraint S3 (Geographical Accuracy) is often only applied to nodes with a degree $\deg(u) \neq 2$ (intersection nodes with $\deg(u) > 2$, and terminus nodes with $\deg(u) = 1$). Line graph nodes of degree 2 are then contracted before schematization. The contracted

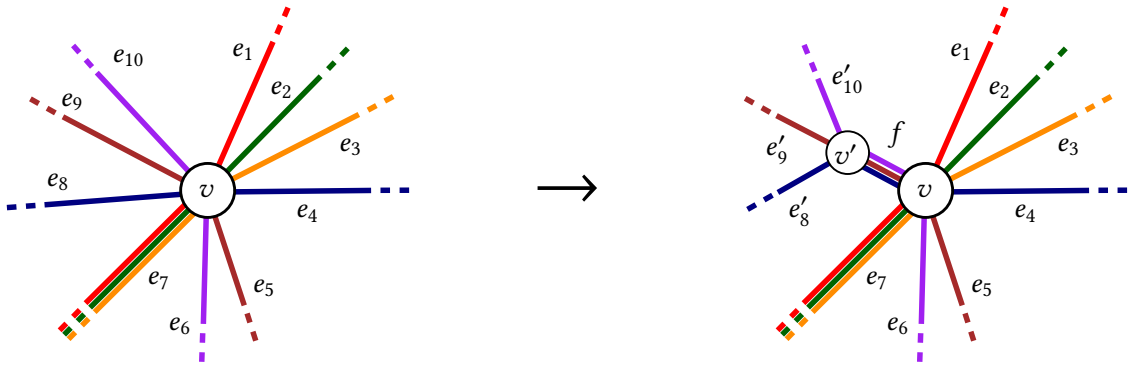


Figure 5.2: Left: Node of an input line graph G with $\deg(v) > 8$. Right: To be able to draw G in an octilinear fashion, we add an additional non-station node v', inflect edges e_8, e_9, e_{10} to v', and connect v and v' by an additional edge f holding the combined lines.

nodes are later re-inserted equidistantly onto the schematic drawing. This greatly reduces the complexity of the problem and has been found to improve the overall map appearance [118]. We call this the degree-2 heuristic. Figure 5.1 gives an example.

5.1.2 Station Labels

As station labels are an essential part of schematic transit maps, labeling is often discussed together with map schematization. Two general approaches exist: *A posteriori* labeling adds station labels *after* the schematic map has been generated. This amounts to basically the same problem as labeling geographically accurate transit maps.

As station labels might require explicit free space around stations or influence the optimal layout, *integrated* labeling optimizes both the labeling and the schematic map *together*. The labels act as rectangular obstacles for the stations and lines.

Labeling was not the focus of this work. A simple baseline approach to a posteriori labeling is described in Chapter 6.

5.1.3 Extending the Maximum Input Node Degree

An input line graph G only allows for an octilinear representation if its maximum node degree is ≤ 8. While real-world public transit networks typically satisfy this constraint, this is not guaranteed. In the course of this chapter we will also introduce several alternative layouts which require a maximum input node degree of 6 or even 4.

To still allow for schematic drawings of input graphs whose maximum node degrees are greater, we use the following approach: Given a node v with $\deg(v) > D$ and incident edges $e_1 \dots e_{\deg(v)}$, as depicted in Figure 5.2. We first add a new non-station node v' to G. Afterwards, for edges $e_{D+1} \dots e_{\deg(v)}$, we substitute v for v'. An additional edge

$f = \{v', v\}$ is then added which holds all lines on the previously inflected edges, that is $L(f) = \bigcup_{e_{D+1} \ldots e_{\deg(v)}} L(e)$. If we now still have $\deg(v') > D$, we repeat to process for v'. Figure 5.2 depicts v' with a different position than v'. In practice we will move v' to the same position as v to avoid distorting soft constraint S3.

5.2 Related Work

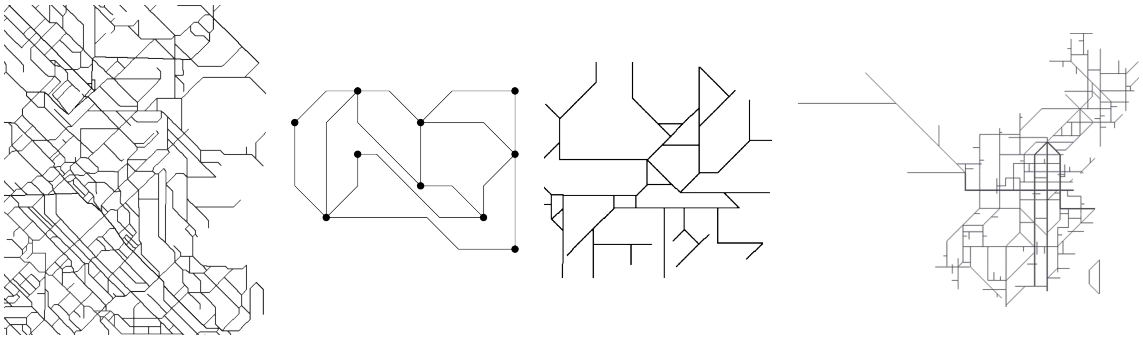
Our work is closely related to previous work on automatic generation of schematic transit maps (often called *metro maps* in the literature), and network schematization in general. We use the terms *metro map* and *schematic transit map* interchangeably from now on. There is also work on the user perception of different layout styles. For an extensive overview and a detailed taxonomization of existing work we refer to a recent survey by Wu et al. [151]. Additionally, there are several classic algorithmic problems that are related to our method of finding metro map drawings on an auxiliary base grid.

5.2.1 Generation of Schematic Maps

A substantial body of work is concerned with the generation of schematic maps in general, often for road network schematization. While these works are all applicable to the problem of generating schematic transit maps (some of them explicitly mention this use case), they often ignore specific aspects of schematic transit maps and typically come without any experimental results on real-world transit networks.

Already in 1988, Elroi [54, 55] suggested three steps for the schematization of road networks: (1) line simplification, (2) line re-orientation along a grid, and (3) enlargement of high density areas. Neyer [111] considered the problem of simplifying a single polyline P into a schematic polyline Q consisting of segments following a predefined set of orientations (e.g. diagonal, vertical, or horizontal). Q should consist of the minimal number of segments such that the Fréchet distance between P and Q is within some ϵ.

Avalar and Müller [12] described an approach to octilinearize road maps by specifying a set of constraints per node (minimum distance to other network nodes, octilinearity of adjacent edges and topology preservation). The network was first simplified using the Douglas-Peucker algorithm. The remaining nodes were then visited iteratively and moved to a nearby position which did not violate any constraints (or left untouched if this was already the case). Although the experiments were done with road networks, the applicability of the approach to transit maps was explicitly noted (but not tested). Final maps were only optimal in the sense that as many nodes as possible did fulfill the constraints, and users had to manually abort the iterative process as soon as they were pleased with the current result. This approach was extended by Ware et al. [144] who introduced a cost function to measure the quality of a map and additionally considered



Avelar and Müller, 2000 [12] Cabello et al., 2001 [33] Ware et al., 2006 [144] Li and Dong, 2010 [101]

Figure 5.3: Example results of previous work on schematic maps, taken from their publications. Avelar and Müller, Ware et al., and Li and Dong considered the schematization of road networks (the examples all schematize different networks). Cabello et al. studied mathematical properties of map schematization and used a toy example.

node displacement and edge rotation difference. Simulated annealing was then used for optimization. This approach was later refined to use gradient descent for optimization by Anand et al. in [9].

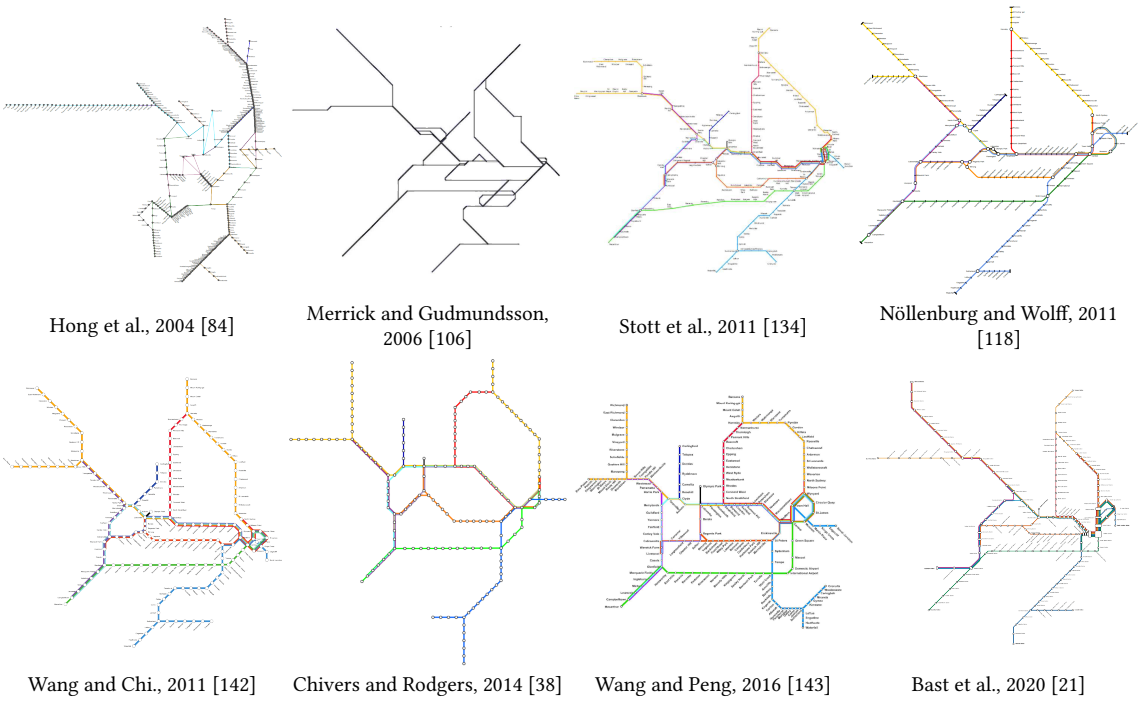
Cabello et al. [33] investigated mathematical properties of drawing octilinear road or railroad maps with at most three segments per edge, while preserving the topology of the input network and maintaining a minimum distance between segments. An $O(n \log^3 n)$ algorithm was described which returned such a schematization if one exists. As in [12], no optimization took place - the returned solution was only guaranteed to be feasible.

In [101], a method for road network schematization based on the concept of strokes (simple paths in the road network) was presented. Several methods to decompose the road network into a set of strokes were presented. The strokes were then simplified using the Douglas-Peucker algorithm and the segments of the simplified strokes were then aligned along an octilinear grid. Afterwards, intersection and terminus nodes were re-projected onto the strokes. An enlargement of high-density areas was later added as a preprocessing step in [137].

Figure 5.3 gives some examples for the approaches described above.

5.2.2 Generation of Schematic *Transit* Maps

The specific problem of drawing *public transit* networks in a way resembling a schematic transit map was first introduced by Hong et al. in [84] and [85]. A set of esthetic criteria was defined and several variations of the classic spring layout algorithm were used to produce schematic graph embeddings resembling a metro map. Regarding topology preservation, it was only required that the resulting drawing should not introduce any edge crossings. Because the geography of the input stations was also ignored, the resulting maps could differ greatly from the topology of the original network. As a prepro-



Hong et al., 2004 [84] | Merrick and Gudmundsson, 2006 [106] | Stott et al., 2011 [134] | Nöllenburg and Wolff, 2011 [118]

Wang and Chi., 2011 [142] | Chivers and Rodgers, 2014 [38] | Wang and Peng, 2016 [143] | Bast et al., 2020 [21]

Figure 5.4: Example results of previous work on octilinear transit maps, taken from their publications.

cessing step, the degree-2 heuristic was introduced (see Section 5.1.1). Station labeling was added a posteriori. Although octilinearity was identified as being a key feature of esthetically pleasing metro maps, the resulting drawings usually did not fulfill this.

Merrick and Gudmundsson [106] built on the work of Neyer [111] and proposed a polyline simplification method specifically designed for metro map generation. The Fréchet distance used in [111] was found to produce unnaturally looking outliers in some cases. They employed the Hausdorff distance instead. To extend this line simplification method to networks, an ordering on the edges of the input network was determined. The corresponding polylines where then simplified in this order. As soon as an edge was simplified this way, the position of its adjacent network nodes was fixed. The schematization of subsequent edges then respected these fixed nodes. The method was tested on multiple real-world public transit networks. As the local simplification did not consider other already simplified input edges, their approach did not guarantee topological correctness.

Stott and Rodgers presented a more refined set of esthetic criteria in [133], thus achieving greatly improved results. They applied local-search techniques to find an optimal metro-map embedding by iteratively moving input nodes on a grid, selecting the most promising new position candidate. In [134], Stott et al. refined this local search by three clustering techniques which grouped nodes that were then moved together (thus

enabling the local search to arrive at neighborhoods behind a local minimum). Labels were optimized together with the layout. As the final drawing was a node embedding (and edges therefore straight lines between nodes on a grid), the main problem of their approach was that the resulting embeddings were not guaranteed to be octilinear. Our method may be understood as an extension of the method by Stott and Rodgers. The major difference is that in our case, edges in the final schematic drawing are shortest paths on an octilinear grid, instead of straight lines. This not only guarantees octilinear results, but has other benefits as well (for example, obstacle avoidance).

Nöllenburg and Wolff described a novel approach in [117]. There, they introduced the hard and soft constraints given in Section 5.1. The goal was to find a node embedding of a given line graph which preserved the topology, respected the original station positions, and consisted of edges drawn as horizontal, vertical, or diagonal lines. Finding such an embedding was shown to be NP-complete by a reduction from Planar 3-Sat in [115]. A mixed-integer linear program (MLP) was described which optimized an objective function built from the soft constraints, subject to the hard constraints. In [118], this MLP was extended to guarantee enough space around station markers for labels. While the resulting maps were esthetically pleasing, it took considerable time to solve the MLPs to optimality (see Table 5.1). Milea et al. [107] added constraints to the original MLP formulation to ensure that the schematic map preserves shortest paths in the input graph.

Wang and Chi [142] described a schematization approach in which the goal was to dynamically focus on routes through the network to ensure readability on mobile devices with small screens. The route focus is optional, though, and the method can be used to generate classic metro maps of high quality. They first compute a curvilinear map which is then refined into octilinearity. The objective function was formulated in terms of multiple energy terms and optimized in a least-squares sense. Reported schematization times were very good, with typical maps being schematized in under 1 second. Labeling was done a posteriori. This was the first work to achieve a performance which allowed for interactive use. It was later refined into a fully interactive editor which allowed map editors to drag stations to specified positions [143].

In [153], Wu et al. built on the MLP by Nöllenburg and Wolff to create maps where a single route is highlighted as a straight horizontal line. They then optimized the placement of large pictorial station labels, placed at the top or bottom of the map and connected to stations via a curve. In [152], the labels were placed inside the map. This was done in three steps: first, an unlabeled layout was produced by an MLP, which was then enlarged until there was enough space to hold the pictorial labels. Afterwards, the layout was tightened again to remove unnecessary white space. Chivers and Rodgers [38] reconsidered the force-based directed approach by Hong et al. [84]. They combined a classic spring embedder to balance edge lengths with magnetic forces to snap the edges into octilinear directions. To avoid conflicting forces, they applied an iterative approach:

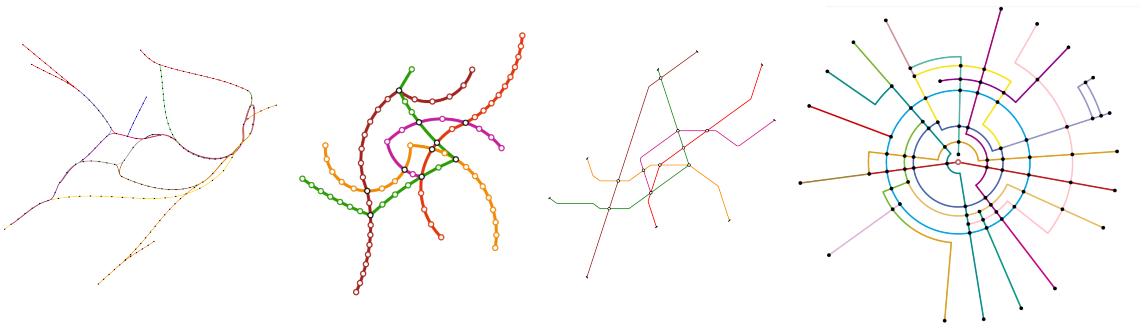
Publication	Network	t
Hong et al. [84]	Sydney	2.3 s
Merrick and Gudmundsson [106]	Sydney	0.3 s
Stott et al. [134]	Sydney	7.2 s
Nöllenburg and Wolff [118]	Sydney	23 m[1]
Wang and Chi [142]	Sydney	0.8 s
Chivers and Rodgers [38]	Sydney	0.6 s
Wang and Peng [143]	Berlin	0.2 s
Bast et al. [22]	Sydney	0.4 s

[1] Suboptimal ILP solution, optimality gap of 16.4% after 12 h

Table 5.1: Reported octilinearization times (all without labeling) from previous work. Wang and Peng also evaluated on Sydney, but without reporting the octilinearization time.

initially, the magnetic forces are given only little weight to produce a balanced, but not octilinear map. They then gradually shift the weight from the spring forces to the magnetic forces with each iteration to arrive at an octilinear result. To further polish the maps, a final post-processing phase is added to straighten peripheral edges. Van Dijk and Lutz [49] described a fast approach to draw a graph such that a given uniform edge length is realized, while the original edge directions should be preserved as good as possible. They formulated a set of constraints (preservation of input node positions and edge length discrepancy) and optimized them in a least-squares sense. An extension of their formulation also tries to optimize for octilinearity. While their approach is very fast (with a typical runtime of under 10 ms), the resulting maps only approximate octilinearity and do not guarantee preservation of topology.

Earlier version of our own approach were described in [21] and [22]. Labeling was done a posteriori. The main difference between our approach and the methods described above is that previous work was interested in octilinear *node embeddings*. Hence, each segment between stations was required to be a straight line (horizontal, vertical or diagonal) in the final drawing, with no edge bends (or the number of bends was fixed to 2, as in [33]). In contrast, our work tries to find octilinear *drawings* where each segment can have an arbitrary number of edge bends. This is motivated by the observation that real-world schematic transit maps are usually not octilinear node embeddings, but octilinear drawings. Our method allows for more flexible layouts. In particular, it is able to route around obstacles like rivers or lakes, or to approximate the real geographical course. We note that Nöllenburg and Wolff [117] already observed that after the degree-2 heuristic was applied and an embedding for the terminus and intersection stations had been found, the final maps had long, unnatural looking straight lines between intersection and terminus stations. Stations were also sometimes heavily displaced to arrive at an embedding. Their solution was to re-add a number of explicit *bend* nodes after applying the degree-2 heuristic and found that 2 bend nodes were enough to produce pleasing results. A similar technique might be used to allow line bends between stations. How-



Fink et al., 2012 [58] van Dijk et al., 2014 [48] Nickel et al., 2019 [112] Niedermann and Rutter, 2020 [113]

Figure 5.5: Example results of previous work on non-octilinear schematic maps, taken from their publications.

ever, the number of required bend nodes is not known beforehand. Additionally, the added bend nodes might unnecessarily elongate line segments because of the minimum segment length required by all of the approaches described above.

5.2.3 Generation of Non-Octilinear Schematic Transit Maps

Several works investigate the problem of drawing non-octilinear schematic transit maps. Fink et al. [58] used a force-based approach to move the control points of cubic Bèzier curves to arrive at curvilinear layouts. They additionally applied checks to ensure that the input topology was maintained. In particular, they strived for a layout where individual transit lines are always drawn as continuous curves, even when they pass through intersections. In [48], van Dijk et al. presented an approach where the network is first decomposed into a set of strokes which are then drawn as circular arcs. Nickel et al. [112] extended the MIP by Nöllenburg and Wolff [118] to be able to render k-linear designs (following angles which are multiples of $360/2k$). For $k = 4$, this is equivalent to rendering octilinear maps.

Since Maxwell Roberts published a map of the London underground network mostly composed of concentric circles (the usability of this layout was later investigated by Newton and Roberts [110]) which received large media attention in Great Britain, orthoradial metro maps have found some research interest. Barth et al. noticed that an orthoradial drawing may be interpreted as a rolled-up orthogonal drawing [13]. Niedermann and Rutter [113] later formulated an ILP to render orthoradial drawings with a minimum number of edge bends and tested their approach on transit maps. Their method did not consider labeling. We will demonstrate that our own approach can also be used to produce orthoradial drawings in Section 5.10. It has to be noted, though, that manually designed concentric circle maps are usually *not* strictly orthoradial (see for example the maps given in [110]). We try to acknowledge this in Section 5.10 where we produce *pseudo*-orthoradial maps.

5.2.4 General Algorithmic Work

A number of classic algorithmic problems are related to drawing (octilinear) transit maps. The rectilinear Steiner minimum tree (RSMT) problem asks to connect a set $S = \{s_1, \ldots, s_n\}$ of points in the plane by a network of vertical and horizontal line segments. The resulting graph may contain additional "bend" nodes where a horizontal segment changes to a vertical segment, or vice versa. The optimal such graph will always be a tree and has a striking resemblance to rectilinear transit maps. The main difference is that input points cannot be moved, and that connections are not part of the input. RSMT is NP-complete [66]. Maurice Hanan introduced the Hanan grid $H(S)$, which is built from a set S of points on the plane by drawing vertical and horizontal lines through each point in S, and adding nodes to their intersections [79]. He showed that for a given set S, $H(S)$ contains an RSMT for S. We will extend this concept to an *octilinear* Hanan grid to reduce the search space size for octilinear transit maps.

Our approach to find schematic drawings by routing input line graph edges through a special base grid bears some resemblance to the k-node-disjoint path problem (NDP). Given a graph G and k node pairs $(s_1, t_1), \ldots, (s_k, t_k)$, NDP asks for a set of k node-disjoint paths $p_1, \ldots, p_k$, where path p_i connects node pair (s_i, t_i) (path start and end nodes do not have to be disjoint). Its NP-completeness was shown by Karp in 1975 [91]. For fixed k, a number of polynomial algorithms exists. Robertson and Seymour showed that, at least theoretically, there is a polynomial-time algorithm for *any* fixed k in the undirected case [126] (for directed graphs, the problem remains NP-complete for any fixed $k > 2$). The general case (were k is part of the input) remains NP-complete for planar graphs [105], grid graphs with rectangular holes [98], and for (solid) grid graphs [97], even if the node pairs are fully disjoint [98]. On grid graphs, the problem was shown to be APX-hard recently [40].

Strongly related to both NDP and our approach is the subgraph homeomorphism problem (SHP). There, the goal is to find homeomorphic images of a pattern graph G in a base graph H. A subgraph homeomorphism may be defined as a pair of mappings $(\mathcal{V}, \mathcal{E})$. $\mathcal{V}$ maps nodes of G to nodes of H. For a node v of G, the mapping $\mathcal{V}(v)$ is called its image. $\mathcal{E}$ maps edges of G to simple paths in H, for an edge e of G, $\mathcal{E}(e)$ is also called its image. For an edge $\{u, v\}$ in G, its image path $\mathcal{E}(\{u, v\})$ is required to start at $\mathcal{V}(u)$ and end at $\mathcal{V}(v)$. Several variants of this problem exist. If the image paths are required to be node-disjoint, and $\mathcal{V}$ is fixed (then called the fixed node-disjoint SHP), the problem is equivalent to NDP. The NP-hardness of the general SHP has been observed early and directly follows from a simple reduction from the Hamiltonian cycle problem [99]. The main difference between SHP and our approach is that our base graph is weighted, and we are not looking for *any* subgraph homeomorphism, but for the optimal one (which minimizes the total sum of edge weights). We also require the subgraph homeomorphism to keep the original circular edge orderings at nodes.

Algorithmic aspects of orthoradial drawings have been investigated in [13] and [114].

In [13], Barth et al. showed that for a given planar input graph, a bend-free planar orthoradial drawing has a combinatorial representation based on the angles of adjacent nodes. Niedermann et al. [114] describe a polynomial-time algorithms to decide whether such a representation is valid, and also describe a polynomial-time algorithm to obtain such a representation. The complexity of finding such a representation with *minimal* number of edge bends was left as an open question.

5.3 Schematization on Octilinear Grid Graphs

As mentioned above, our schematization approach is to find an optimal image of $G = (V, E, L, \mathcal{L})$ in a specially crafted base grid graph. We denote this grid graph by $\Gamma = (\Psi, \Omega)$. The grid cell size is D. Each $v \in V$ is assigned an *image node*, which we denote by $\mathcal{V}(v)$. Each $e \in E$ is assigned an *image path* $p = (\psi_0, \psi_1, \ldots, \psi_{n-1})$ through Γ, which we denote by $\mathcal{P}(v)$. To preserve the topology of the input line graph, we require that for any two input edges e and f, their paths $\mathcal{P}(e), \mathcal{P}(f)$ are node-disjoint (the first and last nodes might be shared if the original input nodes are equivalent). We additionally require that the circular edge ordering of the input graph is preserved. In this section, we describe our general approach by the example of an *octilinear* base grid and show how an optimal such representation can be found both by an ILP (Section 5.4) and a fast approximate approach (Section 5.5). Our goal is therefore to generate octilinear transit maps. We will then extend our approach to other base grids in Section 5.10.

Definition 5.1 (Octilinear Grid Graph)**.** We call a graph $\Gamma = (\Psi, \Omega)$ an octilinear grid graph if it has been constructed as follows: for each position (x, y) on an $N \cdot M$ grid, a grid node $\psi_{x,y}$ is added. For an $\psi_{x,y}$, $N^0(\psi_{x,y})$ denotes the north neighbor $\psi_{x,y+1}$, $N^1(\psi_{x,y})$ denotes the north-east neighbor $\psi_{x+1,y+1}$, and so on. Each $\psi_{x,y}$ is connected by an edge to each of its neighbors $N^0(\psi_{x,y}), \ldots, N^7(\psi_{x,y})$, if they exist.

A simple path $p = (\psi_0, \psi_1, \ldots, \psi_{n-1}), \psi_i \in \Psi$ in such an octilinear grid graph Γ then describes an octilinear curve. Figure 5.6, left gives two examples. To optimize soft constraint S2, we strive for a minimum length of an image path $p(e)$. We therefore only consider *shortest* paths through Γ and assign each edge a *hop cost* w_h. The total cost of a path $p = (\psi_0, \psi_1, \ldots, \psi_{n-1})$ through Γ is then

$$c(p) = (n - 1) \cdot w_h. \tag{5.1}$$

5.3.1 Line Bend Penalties

We again consider Figure 5.6, left. A shortest path in an octilinear grid graph might contain a number of unnecessary bends. For example, the vertical and diagonal segments of path $\{t, u\}$ in Figure 5.6, left can be distributed more optimally: path $\{t, u\}$ in

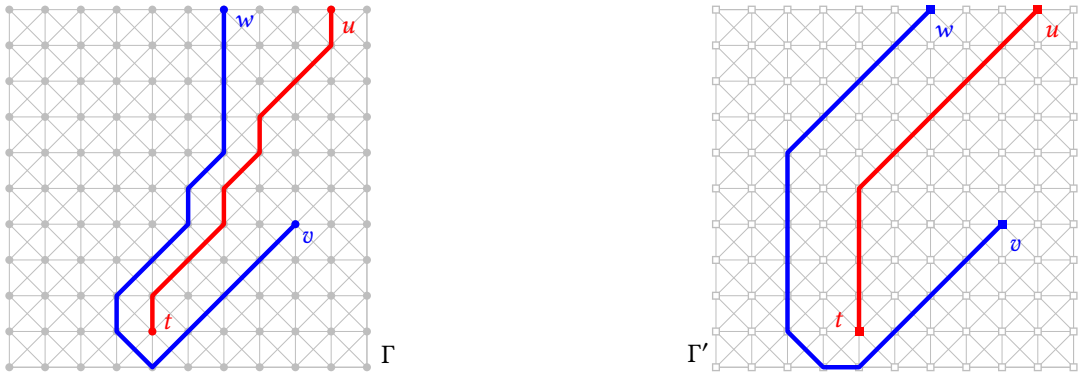


Figure 5.6: Left: Two node-disjoint shortest paths $\{u, t\}$ and $\{w, v\}$ on an octilinear grid graph Γ with uniform weights. Right: Same shortest paths on an undirected octilinear grid graph Γ' extended by explicit bend edges, with higher costs for acute bends.

Figure 5.6, right has exactly the same number of vertical and diagonal edge segments, but also minimizes the number of edge bends required, and uses an obtuse angle at the bottom of the grid. To consider soft constraint S1, we also want path costs to reflect the number and acuteness of bends. Each bend should be punished by a penalty depending on its angle, either w_{135}, w_{90} or w_{45}, with $w_{135} \leq w_{90} \leq w_{45}$ to favor obtuse bends. We do not punish a straight pass through grid nodes and set $w_{180} = 0$. We define

$$B_\psi = \left\{(\psi_1, \psi_2) \in \Psi^2 \mid \exists \psi_0 : \psi_1 \in \text{adj}\,(\psi_0) \wedge \psi_2 \in \text{adj}\,(\psi_0)\right\} \tag{5.2}$$

to hold all node pairs (ψ_1, ψ_2) adjacent to a common node ψ_0 and define the function

$$w_b : B_\psi \mapsto \{w_{180}, w_{135}, w_{90}, w_{45}\} \tag{5.3}$$

to return the bend penalty between edges $\{\psi_1, \psi_0\}$ and $\{\psi_0, \psi_2\}$. We would like our path cost function from above (Equation 5.1) to also consider bends and hence look like this:

$$c_b(p) = c(p) + \sum_{i=1}^{n-2} w_b(\psi_{i-1}, \psi_{i+1}) \tag{5.4}$$

$$= (n-1) \cdot w_h + \sum_{i=1}^{n-2} w_b(\psi_{i-1}, \psi_{i+1}), \tag{5.5}$$

where $n = |p|$ is again the path length and w_h a uniform grid edge weight.

As there are 28 unique edge combinations[1] which produce a bend at a single grid

[1] There are $8 \cdot 7 = 56$ combinations of incoming and outgoing edges (we ignore the case where a path leaves a node through the incoming edge because the path is simple). As our graph is undirected, we consider the bend for edges (e, f) to be equivalent to the bend for edges (f, e).

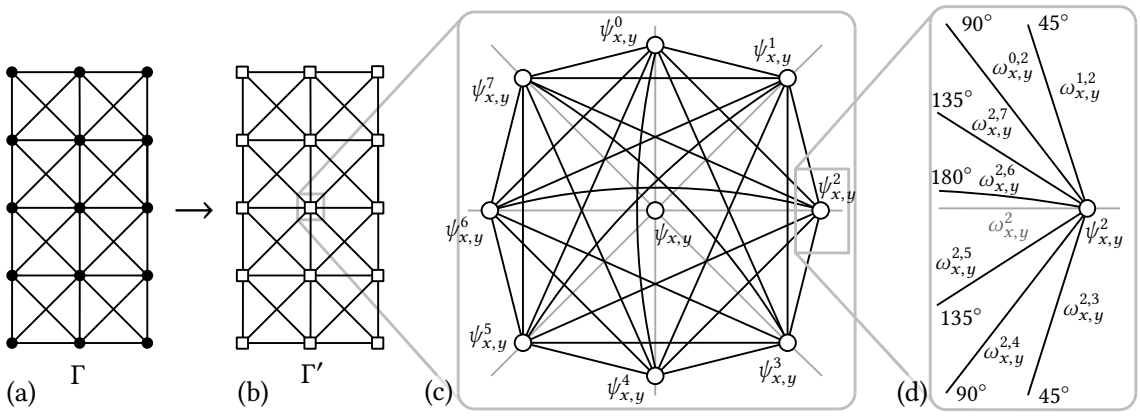


Figure 5.7: Transformation of an octilinear grid graph Γ (a) into its extended version Γ' (b). Each original grid node $\psi_{x,y}$ is extended by 8 port nodes $\psi^0_{x,y}, \ldots, \psi^7_{x,y}$ (c). These are connected to $\psi_{x,y}$ by sink edges $\omega^0_{x,y}, \ldots, \omega^7_{x,y}$. Bend edges $\omega^{i,j}_{x,y}$ (each corresponding to a specific bend angle) connect each port node to its sibling port nodes (d).

node, we cannot model the edge weights in our original grid graph to fit such a cost function. We therefore extend our octilinear grid graph by explicit bend edges.

Consider Figure 5.7. From now on, we denote the set of *original* grid nodes (simply called *grid nodes*) by $\Psi^g \ni \psi_{x,y}$. Correspondingly, the original grid edges (simply called *grid edges*) are denoted by Ω^g. Each such grid node $\psi_{x,y}$ is now extended by 8 so-called *port nodes* $\psi^0_{x,y}, \ldots, \psi^7_{x,y}$, where port node $\psi^i_{x,y}$ leads to the corresponding port node of neighbor $N^i(\psi_{x,y})$. We denote the set of port nodes by Ψ^p. The original edges from $\psi_{x,y}$ to its neighbors are removed. To still be able to reach $\psi_{x,y}$, the port nodes are connected to their grid node by so-called *sink* edges $\omega^0_{x,y}, \ldots, \omega^7_{x,y}$, so $\omega^i_{x,y} = \{\psi^i_{x,y}, \psi_{x,y}\}$. We denote the set of sink edges by Ω^s. Additionally, each port node $\psi^i_{x,y}$ is connected to its clockwise succeeding *sibling* ports $i+1, \ldots, 7-i$ by so-called *bend* edges $\omega^{i,j}_{x,y}$. The set of bend edges is denoted by Ω^b. Each bend edge $\omega^{i,j}_{x,y}$ then corresponds to a traversal of $\psi_{x,y}$ with a bend of either 180°, 135°, 90° or 45°. We call the resulting graph the *extended* grid graph and denote it by $\Gamma' = (\Psi', \Omega')$.

5.3.2 Modeling Edge Weights in the Extended Grid Graph

To arrive at the path cost function from Equation 5.5, the edge weights in Γ' have to be modeled carefully. In particular, we must prevent shortcuts undermining our desired cost model.

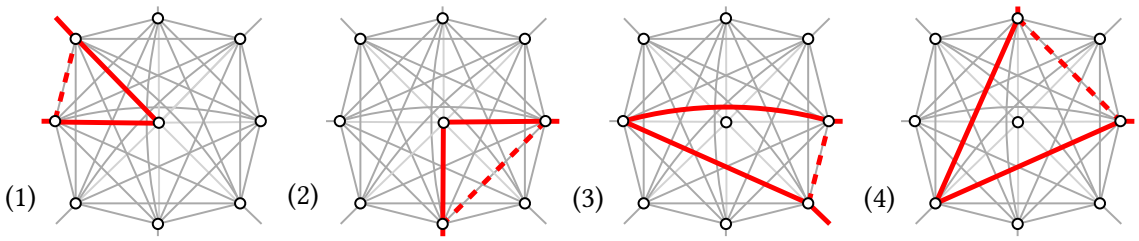


Figure 5.8: Shortcuts in traversing an extended grid node. (1) A 45° bend replaced by two sink edges. (2) A 90° bend replaced by two sink edges. (3) A 45° bend replaced by a 135° bend followed by a 180° bend. (4) A 90° bend replaced by two 135° bends.

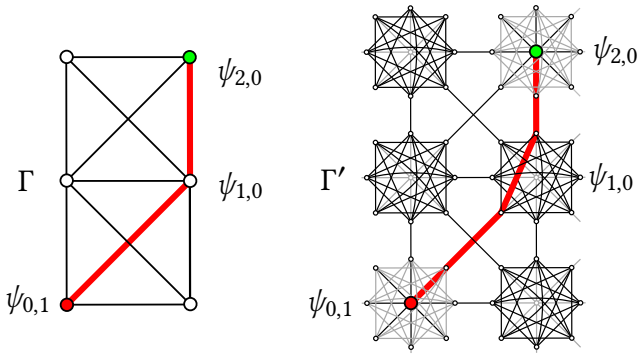


Figure 5.9: Path $(\psi_{0,1}, \psi_{1,0}, \psi_{2,0})$ through the original grid graph Γ and its extended grid graph Γ'. Because sink edges are offsetted by a w_s guaranteed to be greater than any bend cost weight, sink edges will only appear at the beginning and at the end of a shortest path from grid node to grid node.

Sink Edge Weights

A path traversing through a grid node $\psi_{x,y}$ might choose to use two sink edges instead of the corresponding bend edge (Figures 5.8.1 and 5.8.2). To avoid this, we define a uniform sink cost offset w_s in such a way that it is always more expensive than any bend edge. This is e.g. satisfied if we set $w_s = w_{45} + 1$. Given a shortest path p' from a grid node $\psi_{x,y}$ to another grid node $\psi_{x',y'}$ in Γ', it is then guaranteed that only the first and the last edge of p' are sink edges. The first sink edge is used to leave $\psi_{x,y}$, the last sink edge is used to arrive at $\psi_{x',y'}$ (Figure 5.9). Any sink edge appearing in between would compromise optimality, as it could be replaced by a cheaper bend edge.

Bend Edge Weights

Another issue can be seen in Figures 5.8.3 and 5.8.4: in a path traversing through a grid node, both a 45° and a 90° bend edge may be replaced by two cheaper bend edges. A 45° bend edge can be substituted by a 135° bend edge followed by a 180° bend edge (Figure 5.8.3). As we want 180° bends to have no cost, this would essentially mean that any 45° bend may be performed at the same cost as a 135° bend. Analogously, a 90° bend edge may be replaced by two 135° bend edges (Figure 5.8.4).

To block such shortcuts, we add a constant offset $a \geq 0$ to the bend cost and denote these updated bend costs by $w'_{180} = w_{180} + a$, $w'_{135} = w_{135} + a$, $w'_{90} = w_{90} + a$ and $w'_{45} = w_{45} + a$. To ensure the integrity of our original bend costs, we must choose a so that the following inequalities hold:

$$w'_{135} + w'_{180} = 2a + w_{135} + w_{180} \geq a + w_{45} = w'_{45}, \tag{5.6}$$

$$2w'_{135} = 2a + 2w_{135} \geq a + w_{90} = w'_{90}. \tag{5.7}$$

Inequality 5.6 ensures that substituting a 45° bend edge with a 135° bend edge followed by a 180° bend edge is never cheaper than the original 45° bend. Inequality 5.7 ensures that substituting a 90° bend edge with two 135° bend edges is never cheaper than the original 90° bend. As we assume that $w_{180} = 0 \leq w_{135} \leq w_{90} \leq w_{45}$, the inequalities hold for $a = w_{45} - w_{135}$:

$$2(w_{45} - w_{135}) + w_{135} + w_{180} \geq w_{45} - w_{135} + w_{45} \tag{5.8}$$

$$\Leftrightarrow \quad 2w_{45} - w_{135} + w_{180} \geq 2w_{45} - w_{135} \tag{5.9}$$

$$\Leftrightarrow \quad -w_{135} + w_{180} \geq -w_{135} \tag{5.10}$$

$$\Leftrightarrow \quad w_{180} \geq 0, \tag{5.11}$$

and

$$2(w_{45} - w_{135}) + 2w_{135} \geq w_{45} - w_{135} + w_{90} \tag{5.12}$$

$$\Leftrightarrow \quad 2w_{45} \geq w_{45} - w_{135} + w_{90} \tag{5.13}$$

$$\Leftrightarrow \quad w_{45} \geq w_{90} - w_{135}. \tag{5.14}$$

We note that all other possible bend edge replacements (for example, a 90° bend might be replaced by two 45° bends) are already more expensive in our original cost structure, and remain so regardless of the offset a. To also prevent sink edge shortcuts as described above, we have to add a to the previously proposed w_s.

We now again consider a shortest path p' on our extended grid graph Γ' from and to an original grid node. For each $\psi \in \Psi$, we denote by $\psi^* \in \Gamma$ the original grid node ψ belongs to (for port nodes, this is the corresponding grid node, for grid nodes, this is the node itself). The following holds:

Lemma 5.1. *A shortest path $p' = (\psi_0, \psi_1, \ldots, \psi_{n'-1})$ through Γ' with $\psi^*_0 = \psi_0$ and $\psi^*_{n'-1} = \psi_{n'-1}$ (the first and last node of p' are original grid nodes) always describes a simple path $p = (\psi^*_0, \psi^*_2, \psi^*_4, \ldots, \psi^*_{n'-1})$ on Γ, with $|p| = n = |p'|/2 = n'/2$ (for each original grid node visited by p, p' uses two grid nodes).*

Proof. As any $\psi \in \Psi'$ has a corresponding original grid node ψ^*, the existence of a corresponding path p through Γ is easy to see. The start and end node of p will use

the corresponding original grid node and a port node to leave or reach the node. The intermediate grid nodes of p will use a port node to reach the corresponding node in p', and a second port node to leave it. W.l.o.g., we ignore the case where a bend edge is substituted by two or more bend edges of similar cost (as in the examples above), as this neither affects the cost of p', nor the course of the corresponding path p. A shortest path p' through Γ' thus always visits exactly two port or sink nodes belonging to an original grid node, or there is a corresponding shortest path of equivalent cost who does so. As p' is a shortest path, the corresponding path $p = (\psi_0^*, \psi_2^*, \psi_4^*, \ldots, \psi_{n'-1}^*)$ is also simple. □

We now set $w'_h = w_h - a$. Then the following holds:

Lemma 5.2. *Given a shortest path p' through an extended grid graph Γ' with edge weights as described so far. The path p' goes from an original grid node ψ_{x_1,y_1} to an original grid node ψ_{x_2,y_2} and has a corresponding path p in Γ. If the cost $c'(p')$ of p' is optimal (if p' is a shortest path in Γ'), then p optimizes the desired cost function $c_b(p)$ from Equation 5.5.*

Proof. According to Lemma 5.1, such a path p will always exist, and $|p| = n = |p'|/2 = n'/2$. The cost of p' is

$$c'(p') = \overbrace{2w_s}^{\text{sink edges}} + \underbrace{(n-1) \cdot w'_h}_{\text{grid hops}} + \overbrace{\sum_{i=1}^{n-2} a + w_b\left(\psi^*_{2i-2}, \psi^*_{2i+2}\right)}^{\text{offsetted bend costs}} \tag{5.15}$$

$$= 2w_s + (n-1) \cdot (w_h - a) + \sum_{i=1}^{n-2} a + w_b\left(\psi^*_{2i-2}, \psi^*_{2i+2}\right) \tag{5.16}$$

$$= 2w_s + (n-1) \cdot w_h - a + \sum_{i=1}^{n-2} w_b\left(\psi^*_{2i-2}, \psi^*_{2i+2}\right) \tag{5.17}$$

$$= c_b(p) + 2w_s - a. \tag{5.18}$$

Note that w_s and a are positive constants. As a shortest path p' through Γ' thus minimizes $c_b(p) + 2w_s - a$, it also minimizes $c_b(p)$. □

Negatively offsetting the uniform grid cost w_h by a restricts the value range of w_h. Namely, it must hold that $w_h \geq a = w_{45} - w_{135}$, as we would get negative edge weights otherwise. This prevents for example the intuitive cost assignment of $w_h = 1$, $w_{135} = 1$, $w_{90} = 2$ and $w_{45} = 3$. In our experiments, we will always use the cost assignment $w_h >= 1$, $w_{135} = 1$, $w_{90} = 1.5$ and $w_{45} = 2$. Then, $w_h \geq w_{45} - w_{135} = 1$.

5.3.3 Full Optimization Problem

So far, our path cost function c from Equation 5.5 only considers bends that occur along paths between image grid nodes. This ignores bends that may occur between parts of the same line passing through an input node v. To consider this, we also want to minimize the sum c_b^v of bend costs between adjacent image paths $p(e)$ and $p(f)$ where e and f share at least one line (we say $p(e)$ and $p(f)$ are adjacent if they share first or last nodes). If they share more than one line, we multiply the bend cost accordingly. Given for example such paths $p(e) = (\psi_0, \ldots, \psi_n)$ and $p(f) = (\psi'_0, \ldots, \psi'_m)$ with $\psi_0 = \psi'_0$, the bend cost between $p(f)$ and $p(e)$ is $w_b(\psi_1, \psi'_1)$. We can now define the problem of finding an optimal octilinear drawing of G on Γ' as follows:

Definition 5.2 (Optimal Metro-Map Image)**.** Given a line graph G and an octilinear grid graph Γ. For each input node $v \in V$, find an optimal assignment of image grid nodes $\mathcal{I}(v)$ together with an optimal assignment of image grid paths $\mathcal{P}(e)$ for each input edge $e \in E$, if such an assignment exists. We require that image grid paths are non-intersecting and that circular edge orderings at v are preserved in $\psi(v)$ (hard constraint H2). The solution should minimize (1) the sum of all path costs $c_b(\mathcal{P}(e))$, and (2) the sum of the distance $d(v, \psi(v))$ between the original input node v and its image grid node $\psi(v)$, weighted by some w_m (soft constraints S3), and (3) the sum c_b^v as explained above.

More formally, the image $(\mathcal{I}, \mathcal{P})$ of G in Γ should optimize

$$t(\mathcal{I}, \mathcal{P}) = \sum_{e \in E} c(\mathcal{P}(e)) + \sum_{v \in V} d(v, \mathcal{I}(v))\, w_m + c_b^v. \tag{5.19}$$

5.3.4 NP-Hardness

We continue to show that if the costs for edge bends and node displacements are part of the input (as they are in our case), and if obstacles should be considered (which they should in our case), our optimization problem is NP-hard. This proof, which is analogous to the NP-hardness proof of the strongly related subgraph homeomorphism problem given in [99], will serve as the motivation for the ILP described in the next section.

Lemma 5.3. *The decision variant of the Optimal Metro Map Image problem with obstacles is NP-hard.*

Proof. Given a grid graph with holes H with n nodes and m edges. We construct an equivalent octilinear grid graph Γ by setting diagonal edge weights to infinity and adding obstacles corresponding to the holes of H. Horizontal and vertical edge weights are set to 1. Additionally, we set edge bend costs to zero, thus eliminating the need to construct the extended version of Γ. We now construct a graph $G = (V, E, L, \mathcal{L})$ with n nodes, connected in a cyclic fashion. As $\deg(v) = 2$ for any $v \in V$, the circular edge ordering

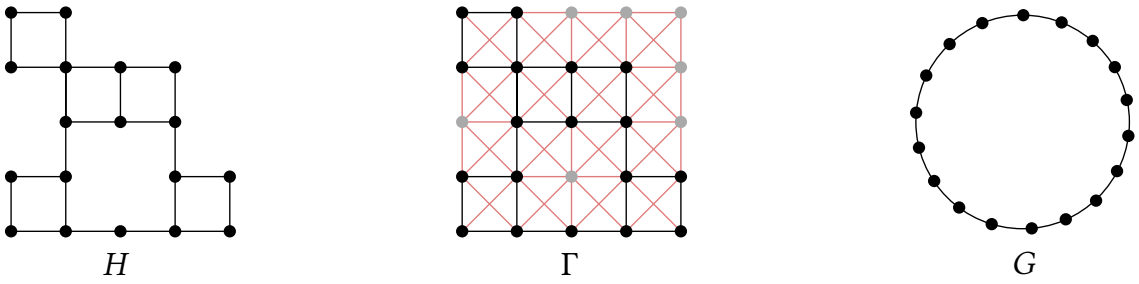


Figure 5.10: The Hamilton path problem on a (general) grid graph H as a special case of our Optimal Metro Map Image problem. A metro map image of G on Γ yields a Hamilton path on H. Red grid edges were outfitted with infinite cost.

at input edges will always be preserved. There is a Hamiltonian cycle in H if and only if there is a metro map image of G in Γ with cost smaller than infinity. As the Hamiltonian cycle problem on grid graphs with holes is NP-hard [88], so is our problem. □

Lemma 5.4. *The decision variant of the Optimal Metro Map Image problem with obstacles is also NP-complete.*

Proof. Given a metro map image, we can obviously calculate its target function value and check whether it satisfies the hard constraints in polynomial time. The Optimal Metro Map Image problem is therefore in NP, and by Lemma 5.3 NP-complete. □

An alternative proof to show the NP-hardness of Optimal Metro Map Image *without* obstacles, but with special node displacement costs (such that grid node images are in effect fixed beforehand) was already mentioned in passing in [21]. It works by reduction from the k-Node Disjoint Paths Problem. Consider the case with zero bend costs, infinite costs for diagonal grid edges, and unit costs for horizontal and vertical grid edges. For a given grid graph Γ and some parameter k, we construct a graph $G = (V, E, L, \mathcal{L})$ with $|E| = k$ and $\deg(v) = 1$ for each $v \in V$ (edges in E are pairwise node-disjoint). Therefore, circular edge orderings at nodes are again always preserved. We choose displacement costs in such a way that each input node $v \in V$ is assigned exactly *one* grid node $\psi_{x,y}$ (e.g. by setting the displacement costs of v to infinity for all other grid nodes). Deciding whether there is an optimal metro map image with cost $\leq m$, where m is the number of horizontal and vertical edges, is then equivalent to the k-Node Disjoint Paths problem on solid grid graphs with disjoint node pairs, known to be NP-complete [98].

5.4 Integer Linear Programming

Motivated by the NP-hardness result from above, we will tackle the Optimal Metro Map Image problem in this section using integer linear programming. Our goal is to optimize

Equation 5.19, while adhering to hard constraints H1-H3. We now use a slightly modified version of our extended grid graph: each (undirected) grid edge $\{\psi, \psi'\}$ is now modeled as a pair of directed edges (ψ, ψ') and (ψ', ψ). For each such edge e (now directed) and each input edge e, we then introduce a binary decision variable $x_{e,\omega}$. $x_{e,\omega}$ should be 1 if ω is a part of the image path $\mathcal{P}(e)$. Each $x_{e,\omega}$ is then added to the objective function with its cost as a coefficient. The rest of this section describes how the hard constraints and the search for shortest paths are modeled. For the latter, we build on the classic linear program formulation of the shortest path problem.

5.4.1 Image Nodes and Station Displacement

To retrieve the image nodes for each input node v, we add a binary decision variable $x_{v,\psi}$ for each pair consisting of an input node $v \in V$ and a grid node $\psi \in \Psi^g$. If $\mathcal{V}(v) = \psi$, then we want $x_{v,\psi} = 1$, or 0 else. As $\mathcal{V}$ is injective, we have to ensure that (1) each input node is assigned exactly one grid node, and (2) each grid node is assigned to at most one input node. To guarantee (1), we add the following constraint:

$$\forall v \in V : \sum_{\psi \in \Psi^g} x_{v,\psi} = 1. \tag{5.20}$$

For (2), we consider the following scenarios: a grid node ψ may either be used as an image for an input node, as a pass-through node on a path, or remain unused. We define Ω^b_ψ to be the set of bend edges belonging to ψ. Then the following set of constraints enforces this:

$$\forall \psi \in \Psi^g : \sum_{v \in V} x_{v,\psi} + \sum_{e \in E} \sum_{\omega \in \Omega^b_\psi} x_{e,\omega} \leq 1. \tag{5.21}$$

If ψ is used as an image for an input node, the first sum is already one. Then, the decision variables $x_{e,\omega}$ for each adjacent bend edge and each input edge must be 0 to fulfill the constraint, making it impossible to reach ψ via a bend edge. Note that in this case, it is still possible to leave or reach ψ via a sink edge. It will become clear below how we ensure that sink edges may indeed only be used if ψ is settled as an image node.

We now consider a grid edge $\omega \in \Omega^g$. Equation 5.21 also ensures that ω may only appear in a single path p through Γ'. Any second path p' using ω would have to pass through the adjacent extended grid nodes, but these are already blocked by p.

5.4.2 Edge Continuity

As each $x_{e,\omega}$ appears in the objective function with its cost as a coefficient, any assignment of $x_{e,\omega}$ to form the path of input edge e is guaranteed to be cost optimal. Let $e = \{s, t\}$. To make sure that the $x_{e,\omega}$ do indeed form a path from $\mathcal{V}(s)$ to $\mathcal{V}(t)$, we have

to enforce that the respective grid edges are connected and that they go from $\mathcal{V}(s)$ to $\mathcal{V}(t)$. To guarantee continuity, it is enough to ensure that each path node except the start and end node has the same number of outgoing and incoming edges. As we ensure that original grid nodes are never part of a path (except the end and start node), we can limit this requirement to port nodes and hence add the following constraints:

$$\forall e \in E \, \forall \psi \in \Psi^p : \sum_{\omega \in \text{adj}^+(\psi)} x_{e,\omega} - \sum_{\omega \in \text{adj}^-(\psi)} x_{e,\omega} = 0, \tag{5.22}$$

To handle start and end grid nodes $\mathcal{V}(s)$ and $\mathcal{V}(t)$, we add the following constraints:

$$\forall e \in E \, \forall \psi \in \Psi^g : x_{t,\psi} - 2x_{s,\psi} + \sum_{\omega \in \text{adj}^+(\psi)} 2x_{e,\omega} - \sum_{\omega \in \text{adj}^-(\psi)} x_{e,\omega} = 0. \tag{5.23}$$

Outgoing edges are now always sink edges, as original grid nodes are not adjacent to other edges in the extended grid graph. The main difference to Equation 5.22 is then that we are counting outgoing edges twice. An outgoing sink edge could now only be regulated by *two* incoming sink edges. But then our path has split somewhere, which is prevented by Equation 5.22. The constraint can thus only be fulfilled if ψ was assigned to the start node s, and thus $x_{s,\psi} = 1$. To counter any incoming sink edge, the only way is to set $x_{t,\psi} = 1$ and settle ψ as the image node for t. As hinted above, this additionally ensures that sink edges may only be used by a path if the corresponding original grid node is settled as the image of either the start or end input node.

Note that Equation 5.22 allows some additional constellations. For example, an outgoing edge may be countered by an incoming edge and by settling ψ as the image of t. While such assignments would be feasible, they would not be optimal (why would we need to leave the target grid node $\mathcal{V}(t)$ again?). However, if we allow grid edge weights of 0, it may be possible for stray circular sub-paths $((\psi, \psi'), (\psi', \psi))$ of cost 0 to appear. We prevent this by the following additional constraints:

$$\forall (\psi, \psi') \in \Omega^g : \sum_{e \in E} x_{e,(\psi,\psi')} + x_{e,(\psi',\psi)} \leq 1. \tag{5.24}$$

Alternatively, such stray circular paths may be removed in a post-processing step.

5.4.3 Topology Preservation

To meet hard constraint H2, we have to ensure that no two image paths intersect and that the circular edge ordering around image nodes $\mathcal{V}(v)$ matches that of the corresponding input node v. Equation 5.21 already ensures that paths never cross at grid nodes, as only a single bend edge may be active per extended grid node, and only a single sink edge may be used at the start and end image node of a path, making it impossible for a path

to arrive and leave via a sink edge.

Because of the diagonal grid edges, however, our octilinear grid graph is not planar. Let Ω^d be the set of diagonal grid edges and for a $\omega \in \Omega^d$, $\omega^\times \in \Omega^d$ is its intersecting diagonal grid edge. To prevent path crossings at such edges, we add the following constraints per diagonal grid edge:

$$\forall \omega \in \Omega^d : \sum_{e \in E} x_{e,\omega} + x_{e,\omega^\times} \leq 1. \tag{5.25}$$

To preserve the original circular edge ordering, we would like to have variables $\delta_{v,e} \in \{0, \ldots, 7\}$ holding the octilinear direction of input edge e at image node $\mathcal{V}(v)$, that is the direction of the corresponding first or last grid edge of $\mathcal{P}(e)$. Consider an input edge $e = \{s, t\}$. At the grid image node $\mathcal{V}(s)$, Equations 5.23 and 5.21 guarantee that only a single outgoing sink edge is used by the image path $\mathcal{P}(e)$. If we only consider the start node s, we can therefore obtain the desired assignment with the following constraints:

$$\forall e = \{s, t\} \in \Omega : \left(\sum_{\psi \in \Psi^g} \sum_{p=1}^{7} p x_{e,(\psi,\psi^p)} \right) - \delta_{s,e} = 0. \tag{5.26}$$

At grid node ψ, an outgoing sink edge (ψ, ψ^p), $p \in 0, \ldots, 7$ adds p to the sum, and as only one such edge is used by $\mathcal{P}(e)$, $\delta_{s,e}$ has to be set to the corresponding octilinear direction to fulfill the constraint. We add a similar constraint for incoming sink edges:

$$\forall e = \{s, t\} \in \Omega : \left(\sum_{\psi \in \Psi^g} \sum_{p=1}^{7} p x_{e,(\psi^p,\psi)} \right) - \delta_{t,e} = 0. \tag{5.27}$$

Let now $u_0, \ldots, u_p, \ldots, u_{\deg(v)-1}$ be the clockwise ordering of nodes u_i adjacent to some $v \in G$. We reconsider the variable $\delta_{v,e}$ defined above. Following an observation made in [115], we note that $\delta_{v,(v,u_p)} < \delta_{v,(v,u_{p+1})}$ must hold for all but one $p \in \{0, \ldots, \deg(v) - 1\}$ if the original circular ordering was kept. We introduce a binary helper variable $\beta_{p,v}$ and add the following set of constraints:

$$\forall u \in V : \delta_{v,(v,u_{p+1})} - \delta_{v,(v,u_p)} + 8 \cdot \beta_{p,v} \geq 1. \tag{5.28}$$

If $\delta_{v,(v,u_p)} < \delta_{v,(v,u_{p+1})}$, the constraint is satisfied. The case were p "carries over" and $\delta_{v,(v,u_p)} > \delta_{v,(v,u_{p+1})}$ is handled by setting $\beta_{p,v}$ to 1. To guarantee that this may only happen once, we add the following simple constraint:

$$\forall u \in V : \sum_{p=0}^{\deg(u)-1} \beta_{p,v} \geq 1. \tag{5.29}$$

We only need to add the constraints from Equations 5.28 and 5.29 to input nodes u with $\deg(u) > 2$. The circular edge ordering of nodes adjacent to 1 or 2 edges is always trivially preserved.

5.4.4 Line Bend Minimization

As explained in Section 5.3.2, line bends along paths are already minimized by our modeling of the edge weights. To also penalize line bends between adjacent input edges in the objective function, we would like to have binary decision variables $\Delta^p_{e,f}$ for each octilinear bend direction p between the image paths $\mathcal{P}(e)$ and $\mathcal{P}(f)$. We already have access to the outgoing and incoming angles of paths $\mathcal{P}(e)$ and $\mathcal{P}(f)$ at their start and end node via the variables $\delta_{v,e}$. For two such variables $\delta_{u,e}$ and $\delta_{u,f}$, $\delta_{u,e} - \delta_{u,f} \mod 8$ is 1 or 7 for 45° bends, 2 or 6 for 90° bends, 3 or 5 for 135° bends and 4 for 180° bends. We denote this value by $\Delta_{e,f}$. Since modulo cannot be part of a linear program, we use the following pair of constraints to get the desired value for $\Delta_{e,f}$. Each constraint is added for each pair of adjacent input edges.

$$\delta_{u,e} - \delta_{u,f} + 8\gamma_{ef} \geq 0, \tag{5.30}$$

$$\delta_{u,e} - \delta_{u,f} + 8\gamma_{ef} \leq 7. \tag{5.31}$$

The auxiliary binary variable γ_{ef} will be 1 if $\delta_{u,f} > \delta_{u,e}$ per Equation 5.30, and only then (Equation 5.31). We therefore get $\Delta_{e,f} = \delta_{u,e} - \delta_{u,f} \mod 8 = \delta_{u,e} - \delta_{u,f} + 8\gamma_{ef}$. As we want to assign different penalties to different bend angles, we introduce 8 binary decision variables $\Delta^0_{e,f}, \ldots, \Delta^7_{e,f}$, one for each possible value of $\Delta_{e,f}$. This assignment is guaranteed by the following constraints, which are again added for each pair of adjacent input edges.

$$\Delta_{e,f} - \sum_{i=0}^{7} i\Delta^i_{e,f} = 0. \tag{5.32}$$

As this might result in multiple incorrectly activated decision variables (e.g. $\Delta^0_{e,f} = 1$ and $\Delta^5_{e,f} = 1$ if $\Delta_{e,f} = 6$), or no activated decision variable at all (if $\Delta_{e,f} = 0$), we additionally ensure that exactly one $\Delta^i_{e,f}$ is activated for each pair of adjacent input edges:

$$\sum_{i=0}^{7} \Delta^i_{e,f} = 1. \tag{5.33}$$

Each $\Delta^i_{e,f}$ is then added with the corresponding bend penalty to the objective function.

5.4.5 Asymptotic ILP Size

We close this section with an analysis of the asymptotic size of our integer linear program. The size is measured in terms of number of constraints and number of variables. Given an input line graph $G = (V, E)$ and a corresponding $X \cdot Y$ grid. Our octilinear grid graph contains $\Theta(XY)$ edges, and $\Theta(XY)$ nodes. As the extended version adds a constant number of edges and nodes to each original grid nodes, the asymptotic number of edges and nodes remains the same. There will be $\Theta(|V|XY)$ station placement variables $x_{e,\omega}$ (one for each pair of input node and original grid node) and $\Theta(|E|XY)$ edge use variables $x_{e,\omega}$ (one for each pair of input edge and extended grid edge). For the circular edge ordering, we introduce two variables $\delta_{v,e}$ for each input edge, so $\Theta(|E|)$ in total, and a constant number of helper variables $\beta_{p,v}$ per input node, so $\Theta(|V|)$ in total. Likewise, we require at most $8 \cdot 7$ helper variables γ_{ef} per input node v for the line bends at input nodes, as there are at most $8 \cdot 7$ pairings of edges per v. Finally, we need 8 bend variables $\Delta^i_{e,f}$ per input node v and edge pair, so at most $8^2 \cdot 7$ per v. The total asymptotic number of variables in our ILP is therefore $O(|E|XY + |V|XY) = O(|E|XY)$.

Regarding constraints, Equations 5.20 and 5.21 add $\Theta(|V|XY)$ constraints. Equation 5.22 adds $8 \cdot 7$ constraints per original grid node and input edge, so $\Theta(|E|XY)$ in total. Likewise, Equation 5.23 adds 8 constraints per original grid node and input edge, so again $\Theta(|E|XY)$ in total. Equation 5.24 adds exactly one constraint per original grid edge, so $\Theta(XY)$ in total. Equation 5.25 adds $\Theta(XY)$ constraints. Equations 5.26 and 5.27 add $\Theta(|E|)$ constraints. Equations 5.28 and 5.29 add $\Theta(|V|)$ constraints. Equations 5.30, 5.31, 5.32 and 5.33 all add at most $8 \cdot 7$ constraints per input node.

The total number of constraints in our ILP is therefore again $O(|E|XY + |V|XY) = O(|E|XY)$. As the grid size $X \cdot Y$ depends on the bounding box area A of the input graph G, both the number of constraints and variables in our ILP is $O(|E| \cdot \lceil A/D^2 \rceil)$.

5.5 Approximate Solution via Iterative Shortest Path Calculation

Our experimental evaluation in Section 5.11 will show impractically high solution times to solve the ILPs to optimality. In some cases, our ILP did not even find a feasible solution in under 12 hours. But even for simple input graphs, the solution times are too long for our ILP to be used in an interactive setting. In this section, we introduce a fast approximate approach that will show to work well in practice. Our basic method consists of 4 steps:

1. Derive an ordering on the input edges E.
2. On the extended grid graph Γ', iteratively calculate the shortest path $\mathcal{P}(e)$ from a set S of start image node candidates to a set T of target image node candidates. We

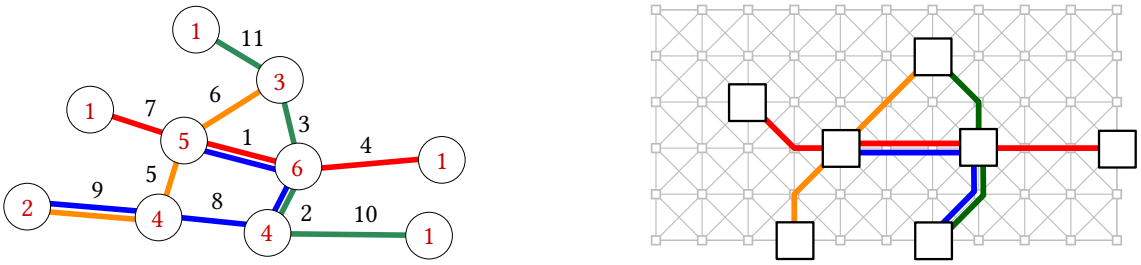


Figure 5.11: Left: Input line graph G with an edge processing order (black) based on the growth method: we select the node with highest line degree (red), process adjacent edges in the order of the adjacent node's line degree and mark the adjacent node as dangling. Then we continue in this fashion from the dangling node of highest line degree. Right: Metro map image after edge 7 has been routed.

call this process *image path routing*. Any path $\mathcal{P}(e')$ which has already been settled should not be crossed. The sink edge weights of S and T reflect the node movement penalty of the respective grid node candidate. We apply some smart heuristics to avoid stalling of our approach.

3. Perform a local search to update node positions and re-route image paths.

5.5.1 Input Edge Ordering

To both find a feasible and good solution, the order in which we process the input edges is crucial, but it is not immediately evident which ordering is best. This section discusses a few methods, which will be evaluated in Section 5.11.

Order by Number of Lines

To settle important network segments first, our baseline idea is to order E in descending manner by the number of lines $|L(e)|$ traversing e. In a typical network, this will lead to central segments being routed first.

Order by Length

As network paths with a high station density are usually the most difficult to octilinearize, we may order E by the length of an edge $e = \{u, v\}$ (induced by the geographical distance between u and v) in ascending manner.

Order by Adjacent Node Degree

Another method of capturing the importance of a line graph edge $e = \{u, v\}$ is the (unordered) pair $\{\deg(u), \deg(v)\}$ of the start and end node degrees. For an edge $e =$

$\{u, v\}$, let $o_{\text{deg}}(e) = (\max\{\deg(u), \deg(v)\}, \min\{\deg(u), \deg(v)\})$. We then say $e >_{\text{deg}} f$ if $o_{\text{deg}}(e) > o_{\text{deg}}(f)$, where $>$ is a lexicographic ordering relation.

Order by Adjacent Node Line Degree

We denote a node's line degree $\text{ldeg}(v)$ to be the number of (non-unique!) lines on each adjacent edge (see Figure 5.11, left, for examples). For an edge $e = \{u, v\}$, let $o_{\text{ldeg}}(e) = (\max\{\text{ldeg}(u), \text{ldeg}(v)\}, \min\{\text{ldeg}(u), \text{ldeg}(v)\})$. We then say $e >_{\text{ldeg}} f$ if $o_{\text{ldeg}}(e) > o_{\text{ldeg}}(f)$, where $>$ again is a lexicographic ordering relation.

Growth-Based

Intuitively, it makes sense to let the map "grow" around a central node. We therefore also consider the following ordering method: First, determine the node v with the highest degree (or line degree). In the order of one of the methods above, add the adjacent edges to the ordered list of edges, add the adjacent nodes to a queue of "dangling" nodes, and mark v as settled. From the list of dangling nodes, retrieve the unsettled node with the highest degree (or, again, line degree) and continue in the same fashion. If no dangling node is left, but unsettled nodes remain, the graph consisted of several connected components, and we continue from the unsettled node with the highest (line) degree.

5.5.2 Edge Routing and Station Placement

After we have obtained an input edge ordering $e_1, \ldots, e_i, \ldots, e_{|E|}$, we iteratively search for a shortest image path $\mathcal{P}(e_i)$ for each $e_i = \{s, t\}$. To also determine the image grid nodes $\mathcal{V}(s)$ and $\mathcal{V}(t)$ at the same time, we search for the optimal set-to-set shortest path from a set S of source grid node candidates s, and a set T of target grid node candidates for t. It can be computed by a variant of Dijkstra's algorithm (see Section 1.3.3 for a more detailed description). The image grid nodes are then settled to the start and end grid nodes of the shortest path for each subsequent iteration. If a later image path is searched for an edge e_j and an adjacent node u has already been settled, then the corresponding candidate grid node set U only contains $\mathcal{V}(u)$. To consider the displacement penalties $d(v, \mathcal{I}(v))\, w_m$ for the image grid nodes, we add the respective displacement cost for s to the sink edges at each $\psi \in S$, and the displacement costs for t to the sink edges at each $\psi \in T$. Note that if S and T overlap, the shortest path (with zero cost) would always be a single node in $S \cap T$. To prevent this, we ensure that $S \cap T = \emptyset$ by first building a Voronoi diagram: for each $\psi_{x,y} \in S \cup T$, move $\psi_{x,y}$ into S' if the geographic distance of its position is closer to the position of s than the position of t. Otherwise, move $\psi_{x,y}$ into T'. Then use S' and T' instead of S and T. In case the image grid node for s or t has already been settled in a previous iteration, we simply remove this grid node from the other candidate set.

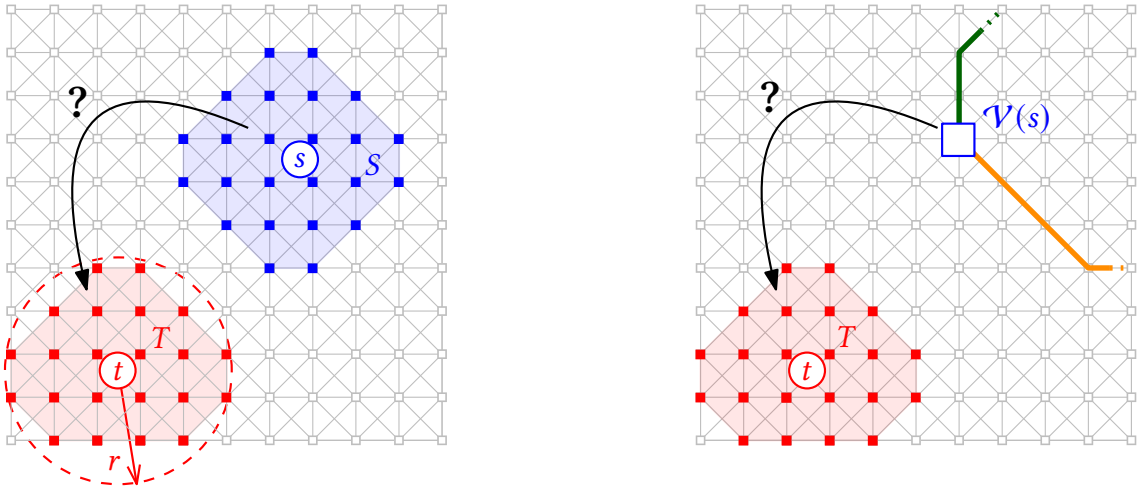


Figure 5.12: Left: To find the image for an input edge $\{s, t\}$ we must find the shortest path from a set of S of grid node candidates for node s to a set T of grid node candidates for node t. We chose the candidates within a distance of r to the original input node position. Right: If one of the nodes (here: $\mathcal{V}(s)$) was already settled, the image search is a one-to-many shortest path calculation from $\mathcal{V}(s)$ to T.

During our experiments, we have found that using all original grid nodes as candidate image nodes for an input node v is excessive, as grid node candidates with high displacement cost are almost never used in the final metro map image. To speed up our approach, we therefore used a radius r in our experiments to limit the size of our image grid node candidate sets S and T (see Figure 5.12, left, for an example). This radius r essentially specifies a maximal node displacement.

5.5.3 Topology Preservation

As in the ILP formulation, we have to ensure that paths do not cross and that circular edge orderings are preserved (hard constraint H2). To achieve this, we update the edge weights of the extended octilinear grid graph after an image path has been found. After such a path $\mathcal{P}(e) = \psi_1, \ldots, \psi_n$ has been found, we set the weight of all bend and all sink edges adjacent to any original grid node ψ_i^* in the path to ∞. We say ψ_i^* is *bend closed* and *sink closed*. This ensures that no original grid node may appear in any image path found in a later iteration. To ensure that image paths are also edge-disjoint, we additionally set the weight of each edge used in $\mathcal{P}(e)$ to ∞.

If a grid node $\psi_{x,y} = \mathcal{V}(v)$ is sink-closed, its sink edges may be opened in a later iteration which searches for an image path for some edge $e = \{v, u\}$. As explained above, the sink edge weights will then be set to the displacement cost of $\psi_{x,y}$ for v. Also note that in such a case, it will still remain bend-closed.

To ensure that paths do not cross at diagonal edges, we use the same technique as

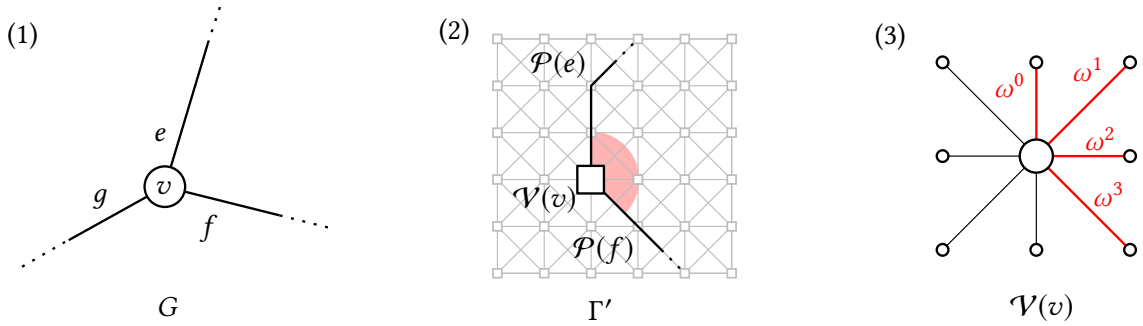


Figure 5.13: (1) An input line graph G. To respect the circular edge ordering around v, we would like to block the area marked red in (2) for routing image path $\mathcal{P}(g)$. We achieve this by setting sink edges corresponding to the octilinear directions of the blocked area to infinity (3).

in the ILP: Let Ω^d be the set of diagonal grid edges and for a $\omega \in \Omega^d$, let $\omega^\times \in \Omega^d$ be its intersecting diagonal grid edge. For each $\omega \in \Omega^d$ which is also part of $\mathcal{P}(e)$, we set the weight of $\omega^\times$ to ∞.

It remains to guarantee that the original circular edge ordering is preserved. Consider Figure 5.13. If for an input edge $e = \{v, u\}$ an image path $\mathcal{P}(e)$ has already been found in a previous iteration, and if we have now found an image path $\mathcal{P}(f)$ for an edge $f = \{v, w\}$ which appears directly before or after e in the original circular edge ordering at v (Figure 5.13.1), we prevent any future image path from violating the original circular edge ordering by closing the corresponding sector between $\mathcal{P}(e)$ and $\mathcal{P}(f)$ (Figure 5.13.2). We do this by setting the weights of the corresponding grid edges to ∞ for any future image path search for an edge adjacent to v (weights $\omega_0 \ldots \omega_3$ in Figure 5.13.3).

5.5.4 Line Bend Minimization

To finally penalize line bends at input nodes, we offset the sink edges at an already settled image grid node $\mathcal{V}(v)$ to reflect the bend costs between the image path $\mathcal{P}(e)$ we are currently searching for and any previously found image paths (Figure 5.14). Each sink edge weight ω_i is offset by the bend cost that would be incurred if sink edge $\omega^i_{x,y}$ was used by $\mathcal{P}(e)$ (and would thus leave in octilinear direction i).

5.5.5 Stalling Prevention Heuristics

During image path routing, locally optimal routing decisions might make it impossible to reach a feasible solution. We apply several locally informed heuristics to prevent such stalling.

We now define the circular distance o between two edges e and f adjacent at v as

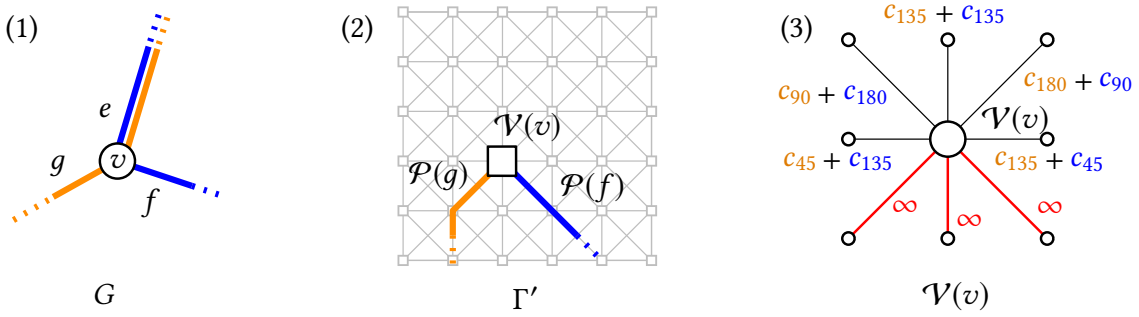


Figure 5.14: (1) Input node v with adjacent edges e, f, g. (2) Grid edge with image paths $\mathcal{P}(g)$ and $\mathcal{P}(f)$ already routed. (3) Sink edge weight offsets at $\mathcal{V}(v)$ prior to routing an image path $\mathcal{P}(e)$ with $e = \{v, u\}$. Sink edge weights reflect the bend penalty between lines on e and lines on g and f. Edges blocked for $\mathcal{P}(e)$ to preserve the topological edge ordering as explained in Section 5.5.3 are marked red.

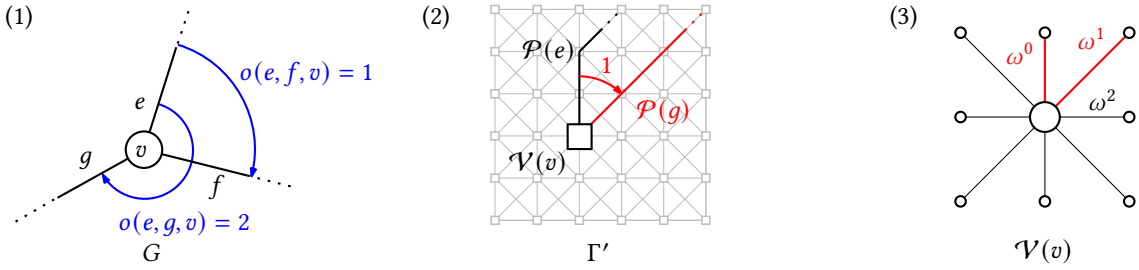


Figure 5.15: (1) A node v with 3 adjacent edges e, f, g for input line graph G. The blue arrows give the circular edge ordering distance for edge e. (2) Image path $\mathcal{P}(e)$ has been routed to leave $\mathcal{V}(v)$ via sink edge ω^0, $\mathcal{P}(g)$ has been routed to leave $\mathcal{V}(v)$ via sink edge ω^1, leaving no space to route $\mathcal{P}(f)$ without violating the circular edge ordering. (3) Sink edges ω^0 and ω^1 are blocked before $\mathcal{P}(g)$ is routed to ensure enough free sink edges.

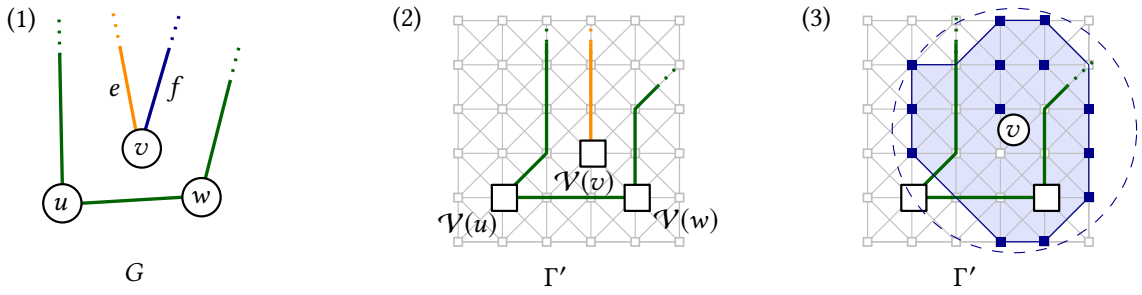


Figure 5.16: (1) Input line graph G. (2) The situation after image path $\mathcal{P}(e)$ has been routed. There is now no path to image node $\mathcal{V}(v)$ which is node-disjoint with already routed image paths. (3) To avoid situations like these, we drop grid nodes with *free* degree lower than $\deg(v)$ as image candidate nodes for v.

the distance between e and f in the clockwise circular edge ordering of v, going in the direction from e to f. For example, if the clockwise circular edge ordering at a node v is e, f, g, h, then $o(e, f, v) = 1$, $o(e, e, v) = 0$, $o(e, g, v) = 2$, $o(e, h, v) = 4$, $o(h, e, v) = 1$, etc. Consider Figure 5.15 and assume the input edge ordering at v has been determined to be e, g, f. Edges e and g have already been routed. In the input graph, the circular distance between g and e is $o(g, e, v) = 2$, and $o(e, g, v) = 1$ (Figure 5.15.1). In the extended octilinear grid graph, the optimal image path $\mathcal{P}(g)$ for g leaves $\mathcal{V}(v)$ at a 45° angle, reducing the circular distance between $\mathcal{P}(g)$ and $\mathcal{P}(e)$ to 1 (Figure 5.15.2). This makes it impossible to route f in a such a way that the original circular edge ordering is respected, as it would have to leave between $\mathcal{P}(e)$ and $\mathcal{P}(g)$ - but there is no grid edge left. To prevent such situations, we ensure that each image path will always leave enough free grid edges in both directions by setting the weight of sink edges corresponding to the octilinear direction which would violate this constraint to infinity (Figure 5.15.2).

Another cause for stalling can be seen in Figure 5.16. Assume the green edges connecting u and w have already been routed, as has the orange edge e connecting v. Routing $\mathcal{P}(e)$ has determined $\mathcal{V}(v)$ to be the optimal grid node for v (Figure 5.16.2). This, however, makes it impossible to route the blue edge f, as $\mathcal{V}(v)$ cannot be reached anymore. To avoid such stalling situations, we exclude grid edges with a *free* node degree smaller than the node degree of input node v from the grid candidates (Figure 5.16.3). The free node degree is the number of adjacent grid nodes that have not yet been blocked.

5.5.6 Complexity

The complexity of our approximate approach basically traces back to $|E|$ shortest path searches through Γ'. To find an initial ordering of the input edges, we have to sort $|E|$ input edges, which can be done in $O(|E| \log |E|)$. To assign each v the set of node candidates with their distance penalties, $O(|V| \cdot |\Psi|)$ distance calculations are required.

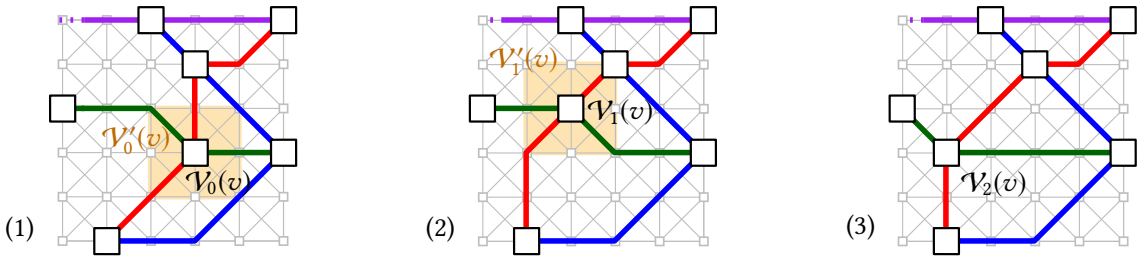


Figure 5.17: Three steps during our local search.

To find the images paths, we require $|E|$ shortest paths searches on the extended grid graph Γ'. The number of nodes $|V'|$ in Γ' is $\Theta(|V|)$, and the number of edges $|E'|$ in Γ' is $\Theta(|E|)$, where V and E are the nodes and edges in the original octilinear grid graph. Using Dijkstra's algorithm, the shortest path searches therefore take $O(|E| \cdot (|\Omega| + |\Psi| \log |\Psi|))$ time. After each Dijkstra run, all grid nodes that appear in the path have to be both bend closed and sink closed. In the worst case, this takes time $O(|\Psi|)$,

Additionally, we have to update the sink edge weights at the image grid nodes after each Dijkstra run. To update the line bend costs, we have to check whether two adjacent edges share a common line. We assume that the lines are given in such a way that we can intersect two sets $L(e)$ and $L(f)$ in time linear in the bigger list (this would be possible if lines are identified by an integer and line sets are sorted.) Then let M again be the total number of unique lines in the line graph. As the maximum input node degree is 8 and the number of line pairs we have to check is upper bounded, the line bend updates per image node take time $O(M)$,

In total, we need $O(|E| \cdot (|\Omega| + |\Psi| \log |\Psi + M))$ time. Just as for the ILP size, $|\Psi| \in O(|\Omega|) = O(XY)$. Therefore, the overall complexity of our approach is $O(|E| \cdot (|\Omega| + |\Psi| \log |\Psi| + M)) = O(|E| \cdot (XY + XY \log XY + M)) = O(|E|XY \cdot (1 + \log XY) + M) = O(|E| \cdot A/D^2 \cdot \log A/D^2 + M)$ (just as for the asymptotic ILP size, A is the area of the input line graph and D is the grid cell size). Just as for the complexity of the edge untangling rules in Chapter 4, M may be considered a constant factor.

5.5.7 Local Search Optimization

Our approximate approach allows for an additional straightforward local search optimization step. Figure 5.17 gives an example. Given an initial feasible metro map image $(\mathcal{V}_0, \mathcal{P}_0)$ on a grid graph Γ. We say the set of all metro map images $(\mathcal{V}'_0, \mathcal{P}'_0)$ where exactly one input node v is moved from $\mathcal{V}_0(v)$ to one of the 8 adjacent grid nodes $N^0(\mathcal{V}_0(v)), \ldots N^7(\mathcal{V}_0(v))$ (if free) and the image paths for all adjacent edges are rerouted is the *search neighborhood* for $(\mathcal{V}_0, \mathcal{P}_0)$. For each neighboring metro map image, we calculate the objective function and take the best $(\mathcal{V}'_0, \mathcal{P}'_0)$ as $(\mathcal{V}_1, \mathcal{P}_1)$.

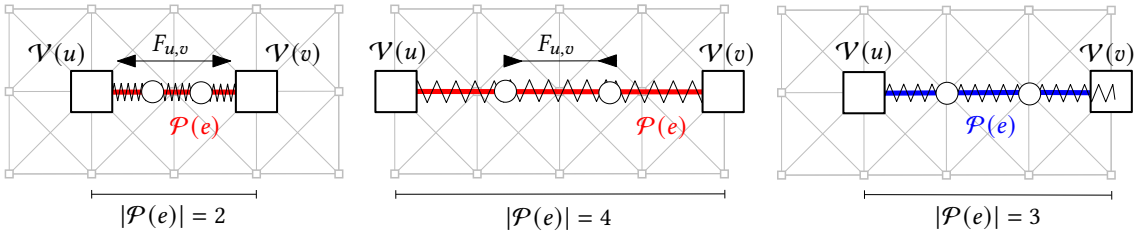


Figure 5.18: Spring-embedding used to enforce a uniform distance between re-inserted nodes during our local-search phase if the degree-2 heuristic was applied. Left: Image path $\mathcal{P}(e)$ is not long enough to hold the $k = 2$ contracted nodes without violating the minimum-distance constraint. The spring stores the potential energy of $E_{u,v} = \frac{1}{4}c$, where c is a penalty parameter. Middle: the edge $\mathcal{P}(e)$ is unnecessarily long, again with a potential energy of $E_{u,v} = \frac{1}{4}c$. Right: $\mathcal{V}(u)$ and $\mathcal{V}(v)$ have been chosen in such a way that the spring is relaxed.

5.5.8 Distance Optimization Between Contracted Degree-2 Nodes

So far, we have ignored that the application of the degree-2 heuristic may lead to very small distances between the re-inserted nodes in the final octilinear drawing. In particular, hard constraint S2 may be violated. While this is sometimes employed in manually created transit maps to save space, extremely small distances between them may lead to overlapping station markers and/or labels. On the other hand, if the distances between them become too great, the resulting lines in the final drawing may appear unnaturally elongated. To address these issues and still keep some flexibility regarding the distance between adjacent degree-2 nodes, we use a spring-based approach. Given an input edge $e = \{u, v\}$ and its image path $\mathcal{P}(e)$. Let $|\mathcal{P}(e)|$ be the number of original grid edges in the path, and let k be the number of nodes on e that have been contracted before schematization. We define a spring force $F_{u,v} = \frac{c}{k} \cdot (k + 1 - |\mathcal{P}(e)|)$ between nodes u and v. The penalty factor c is a parameter of our approach. If $|\mathcal{P}(e)| = k + 1$, we can insert all k contracted nodes in such a way that their positions correspond to an original grid node, and the spring is relaxed. To consider the spring in our local search phase, we add the potential energy $E_{u,v} = \frac{c}{2k} \cdot (k + 1 - l)^2$ to our objective function.

5.6 Speedup Techniques

Apart from general implementation tuning, there are several algorithmic techniques that can be employed to further speed up our approach. This section describes a general A^* heuristic to speed up the image path routing as well as several methods to prune the octilinear base grid. While the former is obviously only applicable to our approximate approach, the latter can also be used to reduce the size and solution times of our ILP.

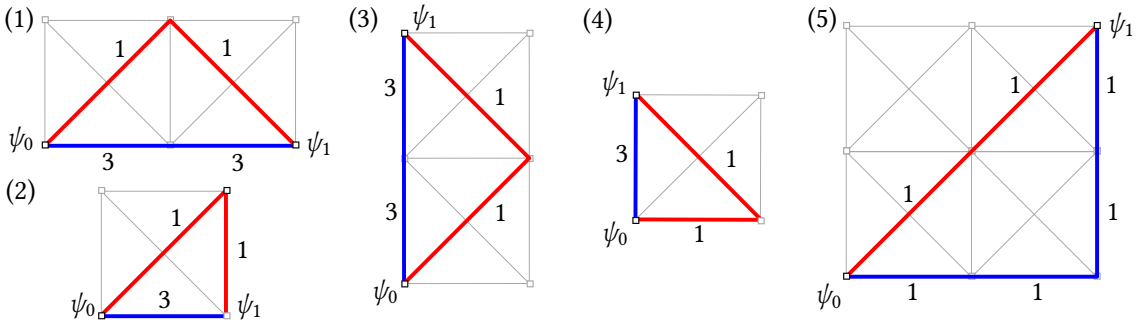


Figure 5.19: (1, 2, 3, 4): If diagonal costs are lower than horizontal or vertical cost, one or two horizontal or vertical edges (blue) may be replaced by cheaper combinations (red) involving a diagonal edge. (5) Example where the Chebyshev distance is a bad heuristic: the shortest path distance between ψ_0 and ψ_1 is 2 (using the diagonal red edges), but the Chebyshev estimates 4 by only considering vertical and horizontal movements (blue).

5.6.1 Shortest-Path Heuristic

In a classic grid graph with diagonal edges and unit edge weights, the L^∞ metric or Chebyshev distance $D_{\text{Ch}}((x,y),(x',y')) = \max\{|x-x'|, |y-y'|\}$ between two grid nodes $\psi_{x,y}$ and $\psi_{x',y'}$ is an admissible A^* heuristic for calculating shortest paths. This is evidently true, as we always need at least $D_{\text{Ch}}((x,y),(x',y')$ edges for a simple path from $\psi_{x,y}$ to $\psi_{x',y'}$. As obstacles may only elongate the shortest path from $\psi_{x,y}$ to $\psi_{x',y'}$, this still holds on octilinear grid graphs with holes or when already routed paths act as obstacles.

From Equation 5.5 and Lemma 5.2, it is clear that $c(p) \leq c_b(p) \leq c'(p)$. Any admissible heuristic for the cost function $c(p)$ is therefore also an admissible heuristic between original grid nodes on the extended grid graph. If we consider unit weights for original grid edges, the Chebyshev distance therefore remains an admissible heuristic there, even if bend and sink costs are considered (intuitively, bend costs may only elongate the shortest path, and sink costs only increase the shortest path costs.) However, we would like our approach to allow for flexible penalties for different bends angles, as well as for using diagonal, horizontal and vertical grid edges. It is clear from above that different bend penalties won't change the admissibility of a Chebyshev distance heuristic.

Let w_h^v be a uniform vertical grid edge weight, w_h^h a uniform horizontal grid edge weight and w_h^d a uniform diagonal grid edge weight. We want our heuristic to reflect these grid edge weights. However, as diagonal, horizontal and vertical edges might be replaced by cheaper combinations of other grid edge types, it is not immediately clear how these weights should be considered. In particular, we want to avoid the trivial admissible method to just multiply the Chebyshev distance by $\min\{w_h^d, w_h^v, w_h^h\}$, as this might result in a bad heuristic. Instead, we use the following approach: For a path from $\psi_{x,y}$ to $\psi_{x',y'}$, we consider two cases: (1) $x = x'$ or $y = y'$, (2) $x \neq x'$ and $y \neq y'$. As a basic "currency", we then use admissible weights $\widetilde{w}_h^h$ and $\widetilde{w}_h^v$. The admissible horizontal

weight $\widetilde{w}_h^h$ is chosen in such a way that for a shortest path from $\psi_{x,y}$ to $\psi_{x',y}$ (note that the y coordinates are the same), $|x - x'| \cdot w_h^h \geq \widetilde{w}_h^h$. Similarly, the admissible vertical weight $\widetilde{w}_h^v$ is chosen in such a way that for a shortest path from $\psi_{x,y}$ to $\psi_{x,y'}$, $|y - y'| \cdot w_h^v \geq \widetilde{w}_h^v$. To find these admissible weights, we have to take shortcuts into account.

For $\widetilde{w}_h^h$, we have to consider two cases: either $w_h^d \geq w_h^h$, or $w_h^d < w_h^h$. In the former case, we can never replace horizontal grid edges by cheaper variants. In the latter case, however, two shortcuts appear: we may either replace two horizontal edges by two cheaper diagonal edges (Figure 5.19.1), or we might replace a single horizontal edge by a diagonal edge followed by a vertical edge (Figure 5.19.2). Note that in both cases, $\widetilde{w}_h^h = w_h^d$ is admissible, regardless of the value of w_h^v. Likewise, $\widetilde{w}_h^v = w_h^d$ is an admissible weight for vertical edges (Figure 5.19.3 and 4). We therefore choose

$$\widetilde{w}_h^h = \begin{cases} w_h^d & \text{if } w_h^d < w_h^h, \\ w_h^h & \text{else,} \end{cases} \tag{5.34}$$

and

$$\widetilde{w}_h^v = \begin{cases} w_h^d & \text{if } w_h^d < w_h^v, \\ w_h^v & \text{else.} \end{cases} \tag{5.35}$$

If for two grid nodes $\psi_{x,y}$ to $\psi_{x',y'}$ we have both $x \neq x'$ and $y \neq y'$, we also have to account for diagonal edges that may be used to replace vertical and horizontal edges (Figure 5.19.5). We additionally define w_h^{-d} to be the weight we might *save* by replacing a vertical and a horizontal edge by a diagonal edge in such a case. This is straightforward:

$$w_h^{-d} = \begin{cases} w_h^d - (\widetilde{w}_h^h + \widetilde{w}_h^v) & \text{if } w_h^d \leq \widetilde{w}_h^h + \widetilde{w}_h^v, \\ 0 & \text{else.} \end{cases} \tag{5.36}$$

We can only replace a vertical and horizontal edge by a diagonal edge if the diagonal weight w_h^d is smaller than $\widetilde{w}_h^h + \widetilde{w}_h^v$. If we do that, we save $\widetilde{w}_h^h + \widetilde{w}_h^v$ at the cost w_h^d. An admissible A^* heuristic for our extended octilinear grid graph with flexible bend penalties and flexible grid edge weights w_h^v, w_h^h and w_h^d is therefore

$$h(\psi_{x,y}, \psi_{x',y'}) = |x - x'| \cdot \widetilde{w}_h^h + |y - y'| \cdot \widetilde{w}_h^v - \min\{|x - x'|, |y - y'|\} \cdot w_h^{-d}. \tag{5.37}$$

In practice, we are searching for the shortest path into a set of target grid node candidates T. The heuristic then becomes $h'(\psi_{x,y}, T) = \min_{\psi_{x',y'} \in T} h(\psi_{x,y}, \psi_{x',y'})$. Although $h(\psi_{x,y}, \psi_{x',y'})$ is easy to compute, we would like to avoid calculating the heuristic for each target node on each Dijkstra relaxation. To reach any node $\psi_{x',y'} \in T$, we must first reach a grid node which is part of the hull of T. The grid node with the minimum heuristic value will be found on this hull. This hull can be quickly precomputed by selecting only grid nodes $\psi_{x',y'} \in T$ which are also adjacent to a node $\psi \notin T$.

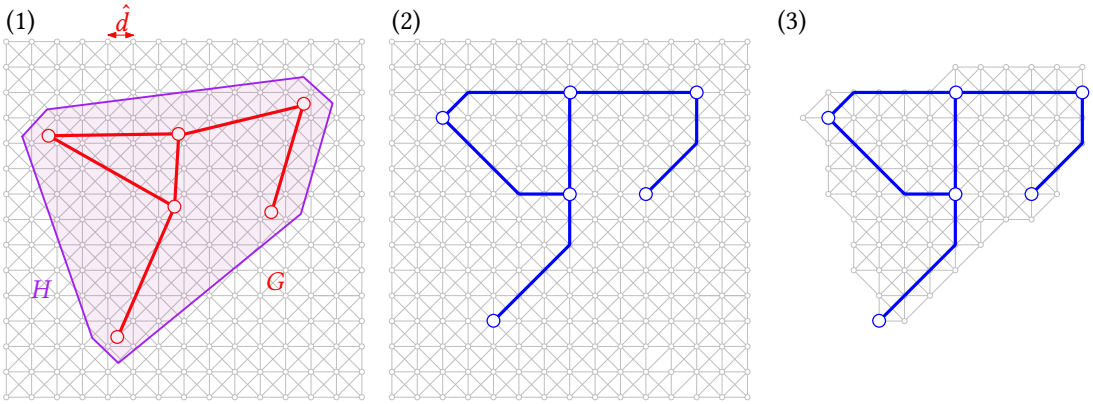


Figure 5.20: (1) Input line graph G on a full octilinear grid graph covering its padded bounding box, and the convex hull H of the input nodes. (2) optimal metro-map image of G on the full octilinear grid. (3) optimal metro-map image of G on an octilinear grid cropped to H.

5.6.2 Sparse Base Grids

As both the asymptotic size of our ILP and the complexity of our approximate approach mainly depend on the base grid size, we would like to study the effects of simplified versions of the octilinear base grid. Figure 5.20 shows an input line graph G, a corresponding extended octilinear grid graph, and the metro map image of G. It is obvious that large parts of the grid graph may be pruned without affecting the metro map image.

In this section, we present three methods to reduce the size of the octilinear grid graph: the first one is straightforward and does not use the bounding box of G to construct the grid graph, but the convex hull. The second approach is based on quadtrees and aims to automatically adjust the grid graph density to the line graph density. The last and most powerful approach extends the concept of a Hanan grid (originally conceived in the context of Steiner trees) to an *octilinear* Hanan grid.

Padded Convex Hull of Input Graph

Let S be the set of coordinates of the input line nodes V. We pad each $p = (x, y) \in S$ by adding 4 additional coordinates $p_1 \ldots p_3$ to S: $p_1 = (x+d, y)$, $p_2 = (x-d, y)$, $p_3 = (x, y+d)$ and $p_4 = (x, y-d)$, where d is a padding parameter. We take the convex hull of P and only keep the grid nodes that are within the hull. These grid nodes already induce an octilinear grid graph with holes, which we use for octilinearization, as shown in Figure 5.20.3. As our method explicitly allows for obstacles, our approach so far (including the A^* heuristic) still works unchanged.

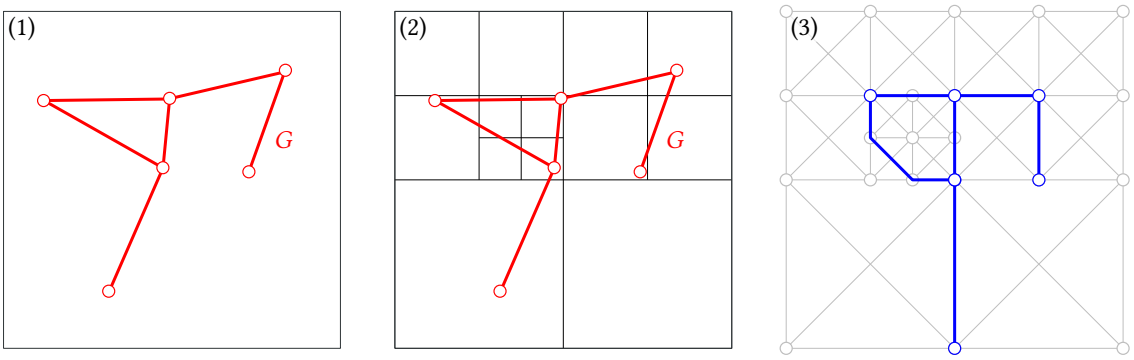


Figure 5.21: (1): An input line graph G and its quadratic padded bounding box. (2): Quadtree constructed from the nodes of G. (3): Octilinear grid graph constructed from the quadtree, and a metro-map image of G in it.

Quadtrees

A quadtree recursively subdivides a rectangle into four smaller rectangles of equal size (quadrants), called cells, until a stopping criterion is met. Given an input line graph G, we build an initial rectangle from the bounding box of G and extend it such that the height and widths are *even* multiples of our grid cell size D (Figure 5.21.1). We then recursively divide the cells until each cell contains exactly one input node (Figure 5.21.2), or until the cell size D is reached. The intersection points of the cell boundaries then act as nodes, and the boundary nodes of each cell are connected by diagonal edges (Figure 5.21.3). This octilinear graph is then extended by bend edges in the same way as our original octilinear grid graph. Grid edges in the resulting octilinear graph may now have lengths which are multiples of the grid edge lengths in the corresponding octilinear grid graph with grid cell size D. To not distort path cost, we multiply the edge weights accordingly.

Octilinear Hanan Grids

Given a set of two-dimensional points S, a Hanan grid $H(S)$ is a graph constructed in the following way: (1) Draw vertical and horizontal lines through each $p \in S$. (2) Where these lines intersect, add nodes. We extend the concept of a Hanan grid by also drawing diagonal lines through each $p \in S$ and call such a grid an *octilinear* Hanan grid.

Definition 5.3 (Octilinear Hanan Grid)**.** Given a point set S on the plane, an octilinear Hanan grid $H^O(S) = (W^O, F^O)$ is the graph constructed from S like follows: (1) Draw vertical, horizontal and diagonal lines through each $p \in S$. (2) Add nodes $w \in W^O$ at intersection points.

Let now S be again the set of input node coordinates. To maintain a minimum segment length in the final map, we have to make sure that edges in $H^O(S)$ have a minimum

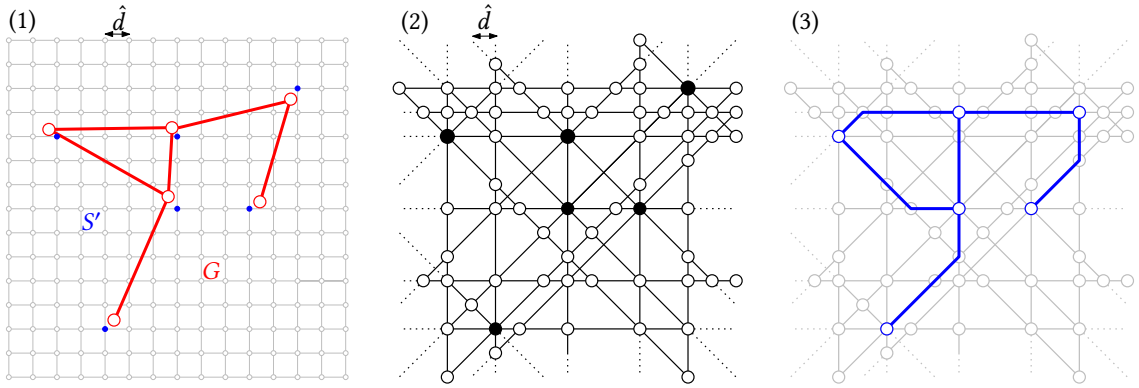


Figure 5.22: (1): An input line graph G on a regular grid covering its padded bounding box. The input nodes are snapped to the nearest grid node and form the input set S' (blue). (2): Octilinear Hanan grid constructed from S'. (3): Metro-map image of G on the octilinear Hanan grid constructed from S'.

length of D. We therefore do not construct the octilinear Hanan grid directly on S. Instead, we first construct a regular grid with cell size D from the (padded) bounding box B of the input graph. Afterwards, we snap each point in S to the nearest grid node (we allow multiple $p \in S$ to share a grid node). We build the octilinear Hanan grid from the resulting point set S'. Additionally, we only keep nodes that fall within B.

For a full octilinear grid graph $\Gamma = (\Psi, \Omega)$ with grid cell size D built from B, $W^O \subseteq \Psi$. However, just like with quadtrees, grid edge lengths may now be multiples of the corresponding grid edge lengths in Γ. We therefore again multiply the edge weights accordingly to preserve path costs. Any path P from s to t in $H^O(S)$ then has a corresponding path in Γ from s to t yielding the same octilinear curve (P may, however, not be the shortest path from s to t in Γ.)

5.6.3 Iterative Octilinear Hanan Grids

An interesting feature of octilinear Hanan grids is that we may increasingly approximate the corresponding full octilinear grid Γ until $H^O(S)$ fully converges to Γ. To achieve this, we first construct an initial octilinear Hanan grid $H^O_0(S)$. We then take the nodes W^O_0 as a point set S_1 and construct an octilinear Hanan grid $H^O_1(S_1)$. This process continues until a maximum number of iterations is reached, or until the grids converge. Figure 5.23 gives an example.

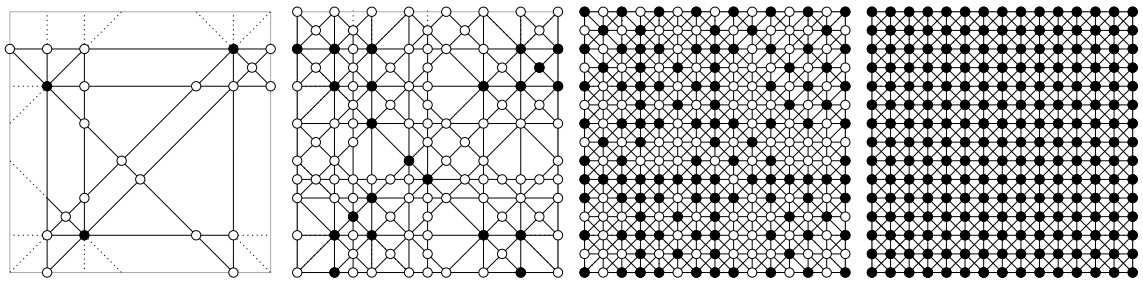


Figure 5.23: Far left: Box-restricted octilinear Hanan grid constructed of 4 input points (black). From second left to right: Octilinear Hanan grids constructed by taking the points of the previous grid as input. After 3 iterations, we arrive at the full grid.

5.7 Obstacles

A common requirement in designing schematic maps is that they should match some existing *base* map. This could be satellite imagery or an existing schematic representation of rivers, lakes, border, coastal lines, forest, or mountain ranges. See for example the excerpt of the official long-distance network map of the Swiss Federal Railways shown in Figure 5.24. As we may add arbitrary offset costs to grid edges, our approach directly allows for a consideration of such obstacles by setting the weight of grid edges intersecting an obstacle to infinity. Figure 5.25 gives an example.

5.8 Approximating Geographical Courses

Another common requirement in schematic maps is that segments should follow their original geographical course to a certain extent, for example if they are used as an overlay for satellite imagery or if large distances between stations would result in an extreme distortion of the course. This becomes particularly important if we apply the degree-2 heuristic prior to the octilinearization. The introduction to this work presented a real-world example of an octilinear map where line segments follow their geographic course between stations (Figure 1.4). If degree-2 nodes were contracted in this example before schematization, most of the lines in this map would appear as mostly straight line segments between intersection or terminus nodes. To approximate geographical courses, a straightforward adaption of our approach is sufficient: instead of using constant grid edge weights, we now use a weight function $w(\omega, e) = w_h + b \cdot d(\omega, e)$, where w_h is the original constant weight for a grid edge, b is a penalty parameter, and $d(\omega, e)$ is the geographical distance between input edge e and grid edge ω.

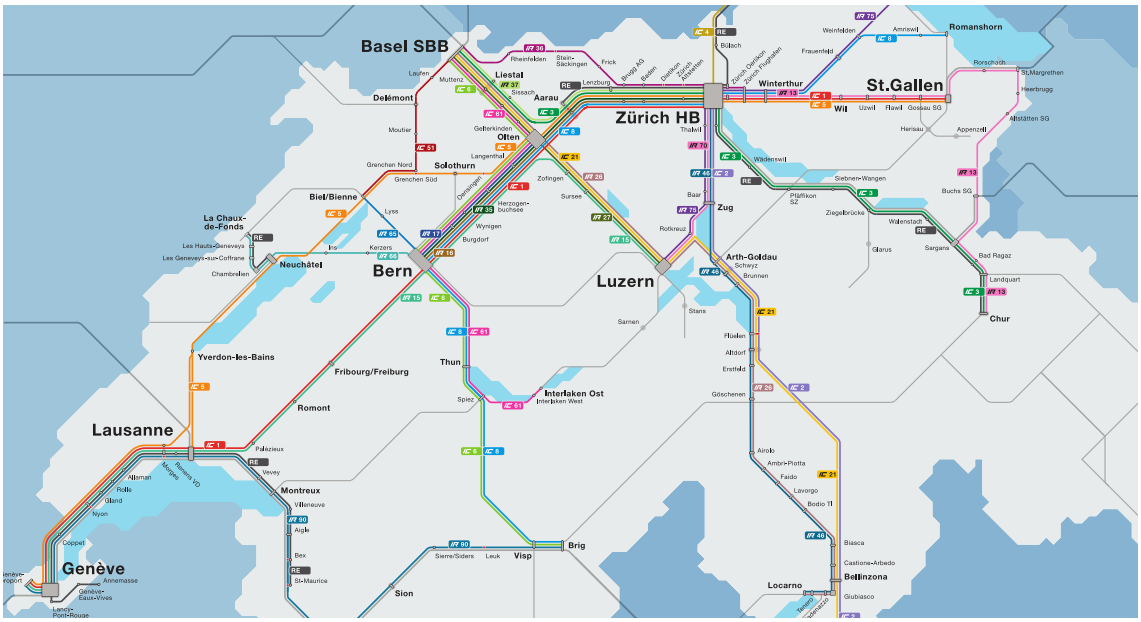


Figure 5.24: Official long-distance network map of the Swiss Federal Railways. The base map depicting schematic representations of the borders and lakes is used in several such maps. The octilinear metro map has to respect these obstacles.

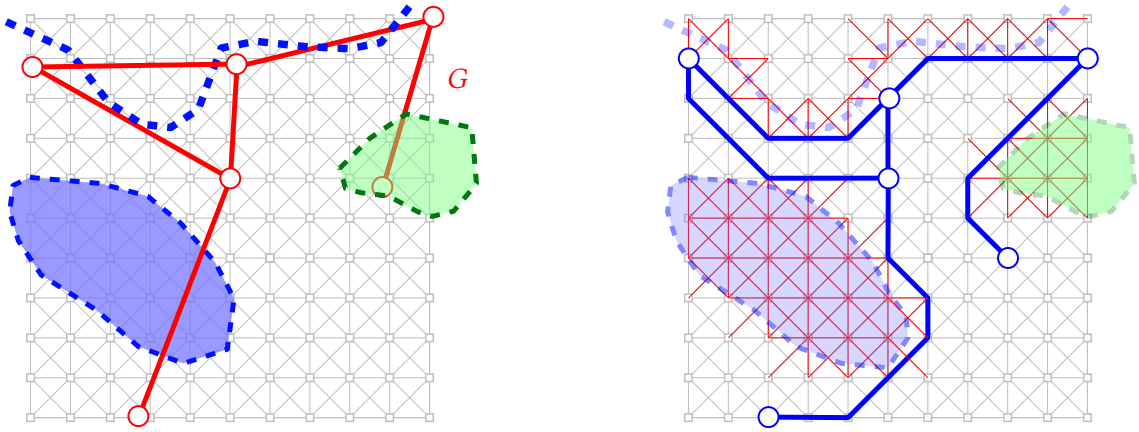


Figure 5.25: Left: A metro-map image for an input line graph G should be found which respects the obstacles formed by a river, a lake (both blue), and a forest (green). Right: Grid edges intersecting the obstacles (red) were outfitted with infinite weight. The resulting metro-map image avoids the obstacles.

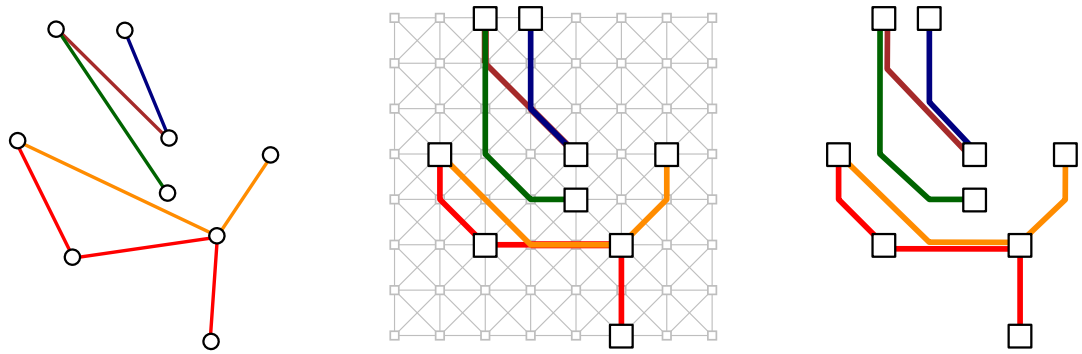


Figure 5.26: Left: Input line graph G. Middle: a metro-map image violating topology constraint H2. Right: The final drawing after additional non-station nodes have been inserted at the positions where the violation first occurred. The octilinearized line graph now allows for a crossing-free line ordering. In the final rendering, the topology is not impaired.

5.9 Hard Constraint Relaxation

Our approximate approach might fail to find an initial feasible metro map image without constraint violations. By relaxing hard constraint H2 we enable our local search phase to better explore the search space. Recall that hard constraint H2 was enforced by closing grid nodes used in a path $\mathcal{P}(e)$ by setting the weight of sink and/or bend edges to infinity. To relax hard constraint H2, we instead introduce a weight w_∞ which is set high enough so that *any* path through Γ using only non-closed edges is cheaper than w_∞. This is for example satisfied if we set $w_\infty = |\Omega| \cdot (\max\{w_h^h, w_h^v, w_h^d\} + \max\{w_{180}, w_{135}, w_{90}, w_{45}\})$. To close an edge in the extended octilinear grid graph, we then use the weight w_∞ instead of ∞. Then, if there are enough grid nodes, we are guaranteed to find an initial metro map image, albeit one that may have topology errors. However, our local search may now find a feasible image respecting hard constraint H2 by trying different grid position.

Interestingly, a violation of hard constraint H2 might not always result in a metro map drawing with incorrect topology. In Figure 5.26, the image paths for the orange and red line, the green and brown line, and the brown and blue line are all not node disjoint and therefore violate hard constraint H2. However, the schematized line graph (with 3 new non-station nodes at exactly those position where the topology constraints where violated) allows for a crossing free line ordering, retaining the topology of the original input graph. The readability of the final map is not impaired by this (implicit) line bundling.

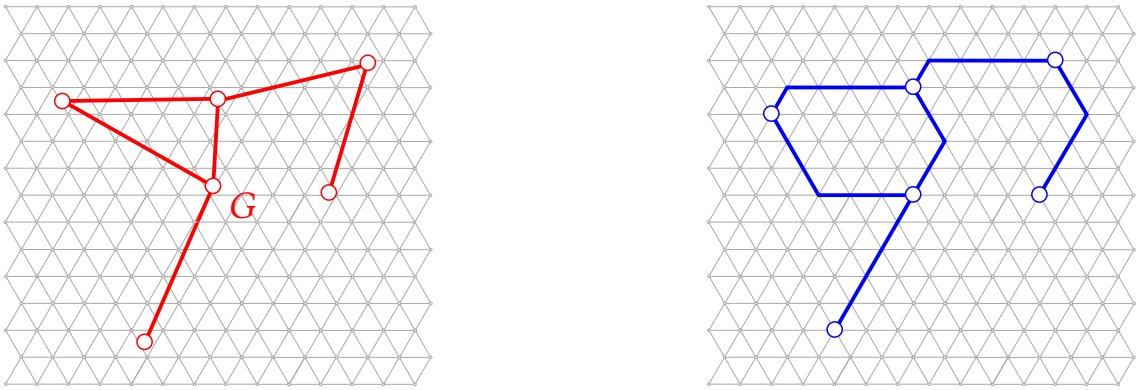


Figure 5.27: Left: An input line graph G and a triangular grid corresponding to the padded bounding box of G. Right: Metro-map image (blue) of G on the triangular grid graph.

5.10 Non-Octilinear Layouts

Except for the A^* heuristic in Section 5.6.1 and the base grid simplification approaches in Section 5.6.2, our approach so far did not require the base grid to be octilinear, or even ortholinear. Both our ILP approach and our approximate approach directly translate to other regular base grids. In this section, we describe two base grids to render hexalinear and orthoradial metro maps.

5.10.1 Hexalinear Maps

In hexalinear metro map drawings, each segment has an orientation which is a multiple of 60°. If we take a triangular grid (a partitioning of the plane into equilateral triangles), any path on the resulting grid will be hexalinear. By using a triangular grid as a base grid we can thus produce hexalinear maps. Adapting the extended nodes described in Section 5.3.2 to the hexalinear setting is straightforward. Just as in the octilinear setting, shortcuts may appear in the expanded grid nodes (which now have 6 ports). We avoid them using the same techniques as described in Section 5.3.2.

5.10.2 Orthoradial Maps

As mentioned in the introduction to this chapter, orthoradial metro maps have found some media and research interest in recent years [13, 110, 114, 113]. A straightforward application of our approach to this setting would be to use an orthoradial grid as a base grid, as depicted in Figure 5.28, left. We call the concentric circles of this grid its *rings*. Such a grid can indeed be interpreted as a rolled-up ortholinear grid as noticed in [13].

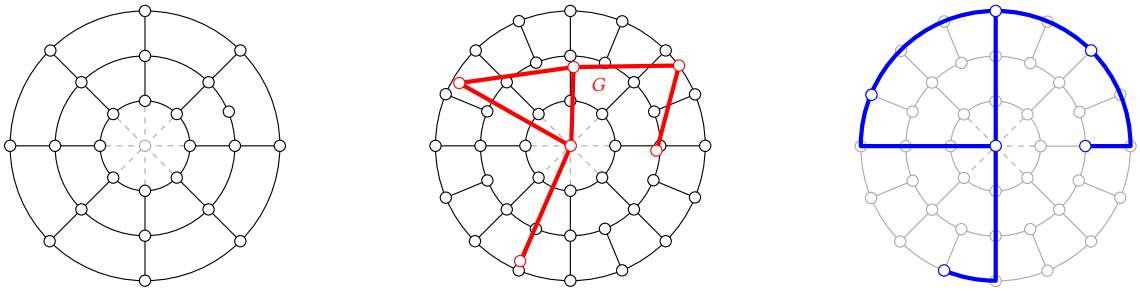


Figure 5.28: Left: An orthoradial grid graph with 3 rings around a center node. With increasing ring radius, the grid density decreases, leaving large empty areas. Middle: A pseudo-orthoradial grid graph in which the the number of nodes on a ring double each time the ring radius doubles, creating a more uniform grid density. Right: A metro-map image (blue) on such a pseudo-orthoradial grid.

From an esthetic point of view though, such a grid is not optimal, as the grid density decreases with the ring radius. This leaves large empty areas in the final map.

Pseudo-Orthoradial Grids

We propose to use a slightly different kind of base grid which we call *pseudo*-orthoradial: starting with an inner ring r_1 of radius d and $b_1 = 8$ equidistant nodes, we keep adding rings r_i with a distance d to the last ring r_{i-1} and nodes at the intersection points of an imaginary line drawn through the center and the corresponding node on r_{i-1} (as in a standard orthoradial grid). However, at rings $r_2, r_4, \cdots, r_{2^j}$, we double the number of nodes, that is $b_2 = 16$, $b_3 = 16$, $b_4 = 32, b_i = 8 \cdot 2^{\lfloor \log_2(i) \rfloor}$. Figure 5.28, middle, gives an example.

The length of segments connecting two rings will obviously always by d. The circular segments are guaranteed to have a length of at least $\frac{\pi}{4}d$. To strictly guarantee a minimum segment length of d on the entire grid, the radius of the inner ring must be set to $\frac{4}{\pi}d$, although we found in our experiments that this hardly matters in practice.

Given an input line graph G, we then chose as a *center* node the node with the highest line degree and use a pseudo-orthoradial grid graph with a radius large enough to cover the entire bounding box of G as a base grid graph. As maps based on concentric circles often feature an additional center node [114], we also add such a node and connect it to the 8 nodes on ring r_1 (see Figure 5.28).

Edge Weights in Pseudo-Orthoradial Grid Graphs

We base the grid edge weights in this pseudo-orthoradial grid graph on a uniform weight for edges connecting two rings, and a base weight w_r for ring segment edges. On the innermost ring r_1, all ring segments are weighted by w_r. For all other rings r_i, we set

Table 5.2: Line graphs used in our experimental evaluation. Under $\bar{d}$ we give the average distance of adjacent input nodes. A is the area of the input graph in km^2. The maximum input node degree is given as $\deg_{max}$. The size of the corresponding grid graph with cell size $D = \bar{d}$ is given in number of nodes $|\Psi|$ and number of edges $|\Omega|$. The last 4 columns give the number of nodes $|V|$ and the number of edges $|E|$ with and without the degree-2 heuristic.

				grid graph		raw		deg-2 heur	
	$\bar{d}$	A	$\deg_{max}$	$\|\Psi\|$	$\|\Omega\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$
Freiburg (FR)	0.4k	75.8	4	3.7k	15.8k	76	79	19	22
Vienna (VN)	0.8k	0.4k	5	5.5k	23.7k	98	104	19	25
Chicago (CG)	1.0k	1.1k	4	10.2k	43.9k	145	151	28	34
Sydney (SD)	1.5k	3.8k	4	14.5k	63.0k	178	183	31	36
Berlin (BE)	0.7k	0.6k	6	10.5k	45.6k	172	184	36	48
Sydney (ST)	0.5k	0.4k	4	12.2k	52.7k	197	205	43	51
London (LO)	1.1k	2.2k	7	18.0k	78.4k	276	313	89	126
New York (NY)	0.6k	1.9k	8	50.4k	221.2k	465	520	113	168

the ring segment weights to $w_r \cdot a$, where a is the ratio between a segment on r_i and a segment on r_1, that is $a = \frac{8i}{b_i}$. As we assume that higher acuteness of a bend angle means higher bend cost, the extended grid nodes will offer no shortcut edges that may circumvent our bend penalty system.

5.11 Experimental Evaluation

We evaluated our approach in all its variants on 8 real-world input line graphs of increasing sizes and complexity: the tram network of Freiburg (F), the subway network of Vienna (V), the light rail network of Chicago (C), the light rail network of Stuttgart (ST), the subway network of Berlin (B), the light rail network of Sydney (SD), the underground network of London (L), and the subway network of New York (NY). For an overview of their dimensions, see Table 5.2.

All but the London subway dataset (which was constructed from data provided in a public GitHub repository[2]) were generated from raw schedule data using the pipeline described so far. Where necessary, we manually fixed small errors in the input data to ensure that the experiments only measured the performance of our schematization technique. For all datasets, we measured the performance of our ILP approach (Sec-

[2] `https://github.com/oobrien/vis/tree/master/tube/data`

tion 5.11.1), the effect of different input edge orderings on our approximate approach (Section 5.11.2), and the quality and speed of the latter (Section 5.11.3). In Section 5.11.4 we also evaluate the effect of the various base grid minimization methods discussed in Section 5.6.2 on quality and speed. In Section 5.11.5, we experiment with base grids for alternative layouts, as described in Section 5.10.

To demonstrate that the resulting schematic maps are visually pleasing, we present various rendered maps for all input graphs. To show the flexibility of our approach, we also experiment with grid edge weights to avoid obstacles, to approximate geographical line courses or to prefer diagonal, horizontal or vertical edges.

Evaluations were run on an Intel Xeon E5649 machine with 12 cores (each with 2.53 GHz) and 96 GB of RAM. All our test datasets as well as the evaluation setup are available online[3]. Our main results can be summarized as follows:

ILPs are too slow for practical use. Even for the simplest input dataset, the ILP took minutes to optimize. For the Chicago dataset, it took 16 hours. For London and New York, no feasible solution was found in under 24 hours.

There is no best input edge ordering for the approximate approach. Under the different edge orderings we tried for the approximate approach, we could find no clear winner.

Approximate approaches are well-suited for practice. For the datasets for which the ILP yielded an optimal solution, the maximum percentage error of the approximate approaches over all tested configurations was only 7.6%. Under visual inspection, we typically saw little to no difference. All but the London and New York dataset were octilinearized in under 1 second.

Sparse grids may speed up the solution, but not always. While the sparse grids reduced the grid graph sizes by up the 82.7% on average, with only little quality loss for the OHG-1 grid, and required less memory for optimization, some of the corresponding ILPs took much longer to optimize. For the approximate approach, however, average solution times decreased by up to 28%.

Hexalinear and orthoradial maps can be quickly rendered using our approach. For all but the London and New York datasets, our approach produced orthoradial layouts in under 1.1 seconds, and hexalinear maps in under 3.3 seconds.

5.11.1 Integer Linear Program

We evaluated our ILP approach on all datasets with the degree-2 heuristic (LP-2), using 3 different grid cell sizes: $D = 0.75 \cdot \bar{d}$, $D = 1.0 \cdot \bar{d}$, and $D = 1.25 \cdot \bar{d}$, where $\bar{d}$ is the average distance between adjacent input nodes. Because the force-based distance optimization penalty described in Section 5.5.8 is quadratic, we could not use it in our ILP approach and therefore could not evaluate LP-2 with additional density penalty. The results for

[3] `https://github.com/ad-freiburg/octi-eval`

Table 5.3: Dimensions, solution times t, and final objective value Θ of our ILPs *with* the degree-2 heuristic (LP-2) for grid cell sizes $D = 0.75 \cdot \bar{d}, D = 1.0 \cdot \bar{d}$ and $D = 1.25 \cdot \bar{d}$. Solution times of — mean no initial feasible solution could be found in under 24 hours. We then give the best bound of the objective function, if one was obtained by the solver.

	$D = 0.75 \cdot \bar{d}$			$D = 1.0 \cdot \bar{d}$			$D = 1.25 \cdot \bar{d}$		
	rows×cols	t	Θ	rows×cols	t	Θ	rows×cols	t	Θ
FR	154.1k×967.3k	5m	0.1k	90.4k×550.1k	3m	0.1k	60.0k×356.5k	1m	93.3
VN	249.2k×1.6M	1h	0.2k	149.4k×934.7k	48m	0.1k	109.5k×672.7k	2h	0.1k
CG	667.8k×4.4M	12h	0.3k	366.5k×2.4M	16h	0.2k	244.7k×1.6M	—	≤0.2k
SD	976.5k×6.5M	5h	0.4k	549.5k×3.6M	47m	0.3k	358.1k×2.3M	3h	0.2k
BE	907.0k×6.0M	2h	0.3k	535.1k×3.5M	58m	0.2k	344.4k×2.2M	—	≤0.2k
ST	1.1M×7.5M	2h	0.4k	658.0k×4.3M	30m	0.3k	423.4k×2.7M	2h	0.3k
LO	4.0M×26.7M	—	≤0.7k	2.3M×15.6M	—	≤0.6k	1.5M×9.9M	—	≤0.3k
NY	15.5M×106.6M	—	—	8.6M×58.7M	—	—	5.5M×37.1M	—	≤0.8k

the ILP sizes (given as number of rows × number of columns), the required optimization time and the final objective value for LP-2 are given in Table 5.3. We also evaluated the ILP approach *without* the degree-2 heuristic in [21], but none of the datasets could be optimized in under 24 hours.

All results in Table 5.3 were computed using gurobi 9.1.2 with default settings (the number of threads was limited to 8 to make the memory consumption manageable), as gurobi proved to be the ILP solver with the best overall performance. Table 5.4 gives a comparison of the GLPK, COIN-OR CBC, and gurobi on all datasets. With GLPK, we were not able to find an optimal solution in under 24 hours for any dataset.

Each dataset was presolved with our approximate approach, and the approximate solution provided to the solver. In [21] it was observed that this warm-start sped up solution times by up to a factor of 2. Note that in the solver comparison in Table 5.4, only GLPK and gurobi were warm-started with the approximate solution, as the COIN-OR CBC solver library offers no straightforward way to provide a first feasible solution.

For grid cell sizes of $D = 0.75 \cdot \bar{d}$ and $D = 1.0 \cdot \bar{d}$, we were able to find optimal solution for all but the London New York dataset. For $D = 1.0 \cdot \bar{d}$, these were typically found in under 1 hour, but for $D = 0.75 \cdot \bar{d}$ and $D = 1.25 \cdot \bar{d}$, optimization took significantly longer. For London and New York, we were not able to find any feasible solution (the solution provided by our approximate approach also had several constraint violations).

In general, the raw ILPs seem to be unsuitable for practical usage, as the optimization takes impractically long, and requires much memory (see Table 5.11 for a comparison).

Table 5.4: Comparison of the performance of 3 different ILP solvers: the GNU linear programming kit (GLPK), the COIN-OR CBC solver (CBC), and gurobi, evaluated on a grid graph with cell size $D = 1.0 \cdot \bar{d}$, and with the degree-2 heuristic enabled. Under t we give the time to optimize the ILP, under *mem* we give the peak memory usage. A t entry of — means that we were not able to find an optimal solution in under 24 hours, A mem entry of — means the optimization was terminated by the out-of-memory killer.

	GLPK		CBC		gurobi	
	t	mem	t	mem	t	mem
FR	—	—	8h	18.5 GB	3m	1.4 GB
VN	—	—	—	31.4 GB	48m	2.9 GB
CG	—	—	—	71.2 GB	16h	38.8 GB
SD	—	—	—	94.7 GB	46m	10.5 GB
BE	—	—	—	91.5 GB	58m	10.5 GB
ST	—	—	—	—	30m	11.1 GB
LO	—	—	—	—	—	—
NY	—	—	—	—	—	—

Moreover, the only solver that was able to find optimal solutions for most of the datasets in under 24 hours was the non-free gurobi solver. However, it seems promising to use the ILPs to optimize the solutions found by our approximate approach for a very short time (e.g. 0.5 seconds) to polish the approximate results. Further experiments are required here, though. In particular, we also have to find a way to incorporate the force-based distance penalty (or a similar technique using linear penalties) into our ILPs.

5.11.2 Input Edge Orderings for Approximate Approach

Table 5.5 gives the final objective functions of our approximate approach with the degree-2 heuristic and *without* the local search phase using 6 different methods for ordering the input edges as described in Section 5.5.1: by number of lines (num lines), by edge length (length), by adjacent node degree (deg), by adjacent node line degree (ldeg), and using the growth-based method with the node degree (gr-deg), or the node line degree (gr-ldeg).

None of the methods consistently outperformed any other method, but with order-by-length (len), there was a clear loser. We also experimented with reversing all 6 orderings or taking a random ordering, but the results were distinctly inferior on all datasets.

For the evaluation of our approximate approach below, we hence always tried *all* 6 methods for finding an initial drawing (this can be easily parallelized). The best solution was then passed to the local search phase.

Table 5.5: Initial results of our heuristic approach *without* the local search phase using 6 different methods of ordering the input edges for rendering: by number of lines on the edge (num lines), by the length of the edge in the input graph, shortest first (length), by adjacent node degree (deg), by adjacent node line degree (ldeg), and growth-based approaches based on the node degree (gr-deg) and the line node degree (gr-ldeg). No clear winner emerged.

	num lines	length	deg	ldeg	gr-deg	gr-ldeg
FR	126.4	128.9	126.1	**118.4**	120.8	118.9
VN	143.9	136.0	**135.3**	135.3	135.7	135.7
CG	$3w_\infty$	**244.6**	$16w_\infty$	$1w_\infty$	251.5	269.7
SD	305.8	**295.1**	306.1	$1w_\infty$	307.6	$1w_\infty$
BE	$3w_\infty$	**259.0**	262.5	270.7	$4w_\infty$	$4w_\infty$
ST	**331.0**	339.1	343.3	334.6	345.2	331.7
LO	$88w_\infty$	$67w_\infty$	$72w_\infty$	$85w_\infty$	$43w_\infty$	$\mathbf{22}w_\infty$
NY	$73w_\infty$	$85w_\infty$	$32w_\infty$	$\mathbf{21}w_\infty$	$35w_\infty$	$60w_\infty$

5.11.3 Quality and Speed of Approximate Approach

Our approximate approach was evaluated both without the degree-2 heuristic (A), and with the degree-2 heuristic (A-2), and with additional density penalty (A-2+D). We focused on 2 aspects: the percentage error δ when compared to the optimal solution obtained via an ILP, and the solution time. As mentioned above, we could not incorporate the quadratic density penalty into our ILP. For A-2+D, we therefore could not measure the approximation error. As we could not find optimal solutions using our ILPs without the degree-2 heuristic, we could also not measure the approximation error for A.

Table 5.6 gives the final objective values, as well as the measured percentage errors for our A-2 on all datasets, when compared to LP-2, again for 3 grid cell sizes: $D = 0.75 \cdot \bar{d}$, $D = 1.0 \cdot \bar{d}$, and $D = 1.25 \cdot \bar{d}$. The solution times and number of local search iterations required for A, A-2, and A-2+D, can be seen in Table 5.7.

The solution quality of the approximate approach A-2 was very good, with average percentage errors for datasets where an optimal solution was available between 1.2% and 3.8% for different grid sizes D. Even for the highly complex London and New York networks, there were only 1-2 constraint violations on a $D = 0.75 \cdot \bar{d}$ grid. However, for larger grid sizes, this increased to up to 16 violations.

The A-2+D approach was able to produce high-quality drawings without topology violations in under 1 second for all datasets except London and New York. Solution times were even lower without the distance optimization between contracted stations (A-2).

Table 5.6: Final objective values of our ILP (LP-2) and our approx. approach (A-2), both with deg-2 heuristic for grid cell sizes $D = 0.75 \cdot \bar{d}, D = 1.0 \cdot \bar{d}$ and $D = 1.25 \cdot \bar{d}$. The percentage error is given as δ. Scores of — mean no solution could be found. If no optimal LP solution was found in under 24 hours, we give the best bound. The average percentage error only includes datasets for which an optimal solution was available.

	$D = 0.75 \cdot \bar{d}$			$D = 1.0 \cdot \bar{d}$			$D = 1.25 \cdot \bar{d}$		
	LP-2	A-2	δ	LP-2	A-2	δ	LP-2	A-2	δ
FR	144.4	147.1	1.9%	115.9	117.0	1.0%	93.3	94.7	1.5%
VN	163.7	164.0	0.2%	128.5	131.7	2.5%	110.8	112.4	1.4%
CG	286.4	290.7	1.5%	229.2	243.8	6.4%	198.0	213.1	7.6%
SD	360.8	363.6	0.8%	286.4	288.8	0.8%	248.0	251.4	1.4%
BE	301.3	304.4	1.0%	238.7	249.7	4.6%	208.0	216.6	4.1%
ST	383.3	391.9	2.2%	308.8	323.1	4.6%	267.2	285.1	6.7%
LO	—	$1w_\infty$	—	—	$16w_\infty$	—	—	$13w_\infty$	—
NY	—	$2w_\infty$	—	—	$7w_\infty$	—	—	$6w_\infty$	—
avg			1.2%			3.3%			3.8%

Table 5.7: Solution times for our linear program and our approximate approach, without (LP, A) and with (LP-2, A-2) the degree-2 heuristic. For the approximate approaches, we also evaluate the running time with an additional density penalty (A-2+D) and give the number of local search iterations until convergence was reached.

	LP	A	its.	LP-2	A-2	its.	A-2+D	its.
FR	—	0.3s	9	3m	75ms	1	0.2s	9
VN	—	0.9s	15	48m	0.1s	5	0.2s	9
CG	—	1.8s	14	16h	0.5s	4	0.7s	11
SD	—	2.2s	13	47m	0.3s	6	0.6s	14
BE	—	3.1s	24	58m	0.4s	4	0.5s	11
ST	—	2.8s	18	30m	0.3s	5	0.7s	18
LO	—	37.3s	72	—	4.4s	15	8.5s	28
NY	—	2m	95	—	34.7s	40	33.7s	40

Table 5.8: Effects of various methods of base grid simplification on base grid graph size, given as number of nodes $|\Psi'|$ and number of edges $|\Omega'|$. Under 'red.' we give the reduction of the number of edges when compared to the full extended grid

	Full	Convex Hull		Quadtree		OHG-1		OHG-2	
	$\|\Omega'\|$	$\|\Omega'\|$	red.	$\|\Omega'\|$	red.	$\|\Omega'\|$	red.	$\|\Omega'\|$	red.
FR	15.8k	7.8k	50.6%	4.6k	71.1%	5.0k	68.7%	15.8k	0.0%
VN	23.7k	10.0k	57.9%	4.5k	80.8%	5.9k	75.3%	23.7k	0.0%
CG	43.9k	17.9k	59.2%	5.6k	87.3%	9.8k	77.6%	43.9k	0.0%
SD	63.0k	40.0k	36.5%	8.0k	87.2%	15.0k	76.1%	63.0k	0.0%
BE	45.6k	25.2k	44.7%	7.9k	82.7%	15.1k	66.9%	45.6k	0.0%
ST	52.7k	30.3k	42.5%	10.4k	80.3%	19.9k	62.2%	52.7k	0.0%
LO	78.4k	47.3k	39.7%	14.4k	81.6%	34.7k	55.8%	78.4k	0.0%
NY	221.2k	92.8k	58.0%	20.3k	90.8%	73.3k	66.9%	221.2k	0.0%
avg			48.6%		82.7%		68.7%		0.0%

5.11.4 Effect of Sparse Base Grids

For the evaluation of the sparse octilinear base grids, our interest was twofold: First, we wanted the know the additional impact sparse base grids have on the solution quality. Second, we were interested in the gained solution time speedup, and memory consumption reduction. As mentioned above and visible in Table 5.4, the ILP optimization required significant memory, making the latter particular important.

The effect of the sparse grid techniques on the dimensions of the underlying grid graph can be seen in Table 5.8. Quadtree based grids reduced the number of edges by 82.7% on average, while the grids based on the convex hull of the input graph already yielded an average reduction of 48.6%. A single-iteration octilinear Hanan grid (OHG-1) gave an average reduction of 68.7%. For all datasets, the octilinear Hanan grid already converged to the full grid after a second iteration (OHG-2), therefore resulting in no reduction.

The impact on the solution quality depended on the type of sparse base grid. For the convex hull based approach and octilinear Hanan grids (OHG-1), the average impact on the solution quality of the ILP based approach was negligible: the average percentage error was 1.4% for the convex hull based approach, and 1.9% for OHG-1 (Table 5.9). Figure 5.29 gives a visual example of a map rendered on a full grid, and various sparse grids. For the quadtree based approach, the average percentage error was much higher at 9.5%, and the ILP for the Chicago network became infeasible. Contrary to our initial hope, the reduced ILP sizes only sometimes resulted in an optimization speedup. For example, al-

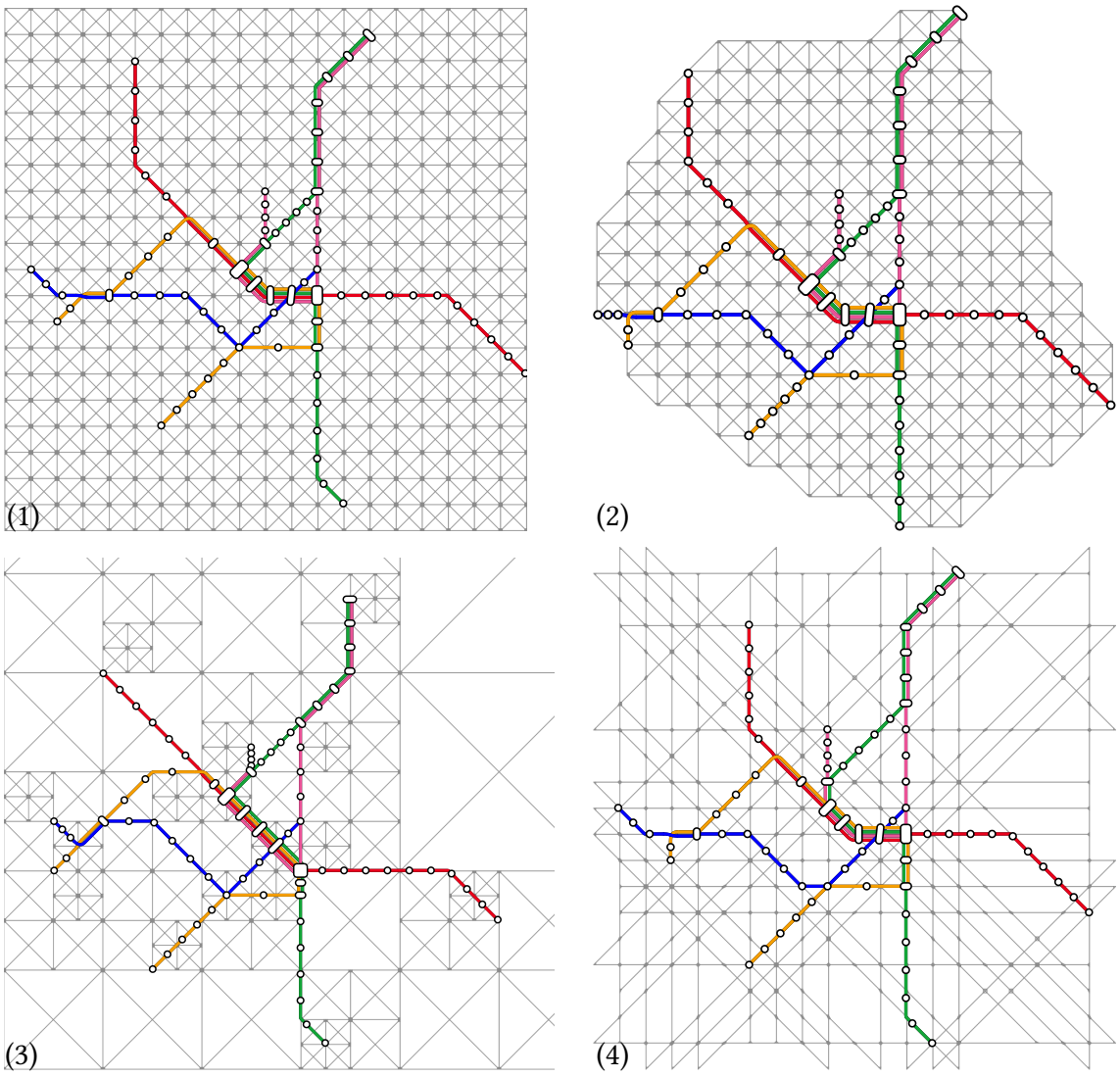


Figure 5.29: 2020 Freiburg tram network, octilinearized using an ILP on (1) a full grid, (2) a convex hull based grid, (3) a quadtree based grid, and (4) an octilinear Hanan grid.

Table 5.9: Effects of various methods of base grid simplification on ILP sizes, solution times and optimality. ILPs were optimized using gurobi.

	Full		Convex Hull			Quadtree			OHG-1			OHG-2		
	Θ	t	Θ	δ	t	Θ	δ	t	Θ	δ	t	Θ	δ	t
FR	115.9	3m	116.8	0.8%	2m	122.1	5.4%	54.0s	117.8	1.6%	53.8s	115.9	0.0%	3m
VN	128.5	48m	129.4	0.7%	27m	145.6	13.3%	42m	130.9	1.9%	38m	128.5	0.0%	52m
CG	229.2	16h	238.8	4.2%	19h	—	—	—	234.3	2.2%	6h	229.2	0.0%	13h
SD	286.4	47m	287.1	0.2%	41m	306.3	6.9%	52m	294.3	2.7%	54m	286.4	0.0%	1h
BE	238.7	58m	241.5	1.2%	5h	271.4	13.7%	13h	243.0	1.8%	6h	238.7	0.0%	1h
ST	308.8	30m	312.5	1.2%	4h	334.3	8.3%	7h	312.6	1.2%	28m	308.8	0.0%	37m
LO	—	—	—	—	—	—	—	—	—	—	—	—	—	—
NY	—	—	—	—	—	—	—	—	—	—	—	—	—	—
avg				1.4%	158.4%		9.5%	495.3%		1.9%	71.7%		0.0%	11.1%

though a single-iteration octilinear Hanan grid (OHG-1) sped up the solution times for Freiburg, Vienna, and Chicago, the time required to optimize the Sydney dataset was significantly longer, and for the Berlin network, it went up from 58 minutes to 6 hours. On average, the solution times hence *increased* by up to 495.3% for the quadtree based approach. In hindsight, this was to be expected, though: the sparse base grids not only make finding an initial feasible solution much harder, they are also much less regular than a full grid.

The speedup was more pronounced for the approximate approach (Table 5.10), with an average reduction of the required solution time by up to 28% for OHG-1. The approximation error, when compared to the approximate approach on the full grid (not to the optimal ILP solution!), was similar to the ILPs, and was also lowest for the convex hull based approach, and highest for the quadtree based grid. Interestingly, the OHG-2 approach produced slightly better results than the full grid, although the grids had exactly the same nodes and edges. The building process of the graphs, however, greatly differs, and the different orderings of the graph adjacency lists resulted in slightly different solutions produced by the initial runs and the subsequent local search.

As expected, the memory consumption of both the ILP and the approximate approach was often significantly reduced (Table 5.11), but in some cases, the ILP solve consumed more memory on the sparse grid than on the full grid (this was for example the case for Sydney on an OHG-1 grid). We suspect a similar reason as for the increased optimization times. Regarding memory consumption, solution time, and solution quality, a single-iteration octilinear Hanan grid (OHG-1) seems to be a good compromise.

Table 5.10: Effects of various methods of base grid simplification on solution times and optimality of our approximate approach.

	Full		Convex Hull			Quadtree			OHG-1			OHG-2		
	Θ	t	Θ	δ	t	Θ	δ	t	Θ	δ	t	Θ	δ	t
FR	117.0	75ms	117.9	0.7%	52ms	133.9	14.5%	0.1s	120.2	2.7%	95ms	117.0	0.0%	82ms
VN	131.7	0.1s	132.9	0.9%	0.2s	152.2	15.5%	0.1s	134.1	1.8%	0.1s	129.4	-1.7%	0.1s
CG	243.8	0.5s	$4w_\infty$	—	0.3s	$2w_\infty$	—	0.3s	$1w_\infty$	—	0.4s	242.9	-0.4%	0.5s
SD	288.8	0.3s	289.0	0.1%	0.3s	312.9	8.3%	0.3s	302.4	4.7%	0.3s	288.8	0.0%	0.4s
BE	249.7	0.4s	252.5	1.1%	0.3s	286.5	14.7%	0.3s	261.2	4.6%	0.2s	249.7	0.0%	0.4s
ST	323.1	0.3s	328.0	1.5%	0.3s	$1w_\infty$	—	0.7s	325.9	0.9%	0.2s	318.1	-1.6%	0.3s
LO	$16w_\infty$	4.4s	$16w_\infty$	—	2.2s	$61w_\infty$	—	1.8s	$18w_\infty$	—	2.4s	$18w_\infty$	—	5.4s
NY	$7w_\infty$	34.7s	$6w_\infty$	—	8.0s	$45w_\infty$	—	9.0s	$8w_\infty$	—	7.4s	$7w_\infty$	—	28.3s
avg				0.9%	-25.8%		13.3%	-2.2%		2.9%	-28.0%		-0.6%	1.7%

Table 5.11: Memory consumption on sparse base grids of our ILP (LP-2) and our approximate approach (A-2), both with the degree-2 heuristic enabled.

	Full		Convex Hull		Quadtree		OHG-1	
	LP-2	A-2	LP-2	A-2	LP-2	A-2	LP-2	A-2
FR	1.4 GB	26.1 MB	795.3 MB	18.0 MB	896.1 MB	16.8 MB	985.2 MB	20.8 MB
VN	2.9 GB	33.3 MB	5.0 GB	21.1 MB	3.0 GB	17.4 MB	3.9 GB	23.2 MB
CG	38.8 GB	52.3 MB	20.5 GB	29.2 MB	—	19.0 MB	8.0 GB	33.5 MB
SD	10.5 GB	68.8 MB	6.8 GB	48.2 MB	8.9 GB	23.4 MB	14.0 GB	44.0 MB
BE	10.5 GB	53.4 MB	5.3 GB	34.9 MB	10.5 GB	22.4 MB	8.6 GB	39.9 MB
ST	11.1 GB	59.7 MB	6.8 GB	39.2 MB	10.7 GB	26.3 MB	4.7 GB	48.3 MB
LO	—	85.4 MB	—	56.8 MB	—	31.7 MB	—	71.4 MB
NY	—	220.1 MB	—	100.7 MB	—	43.2 MB	—	149.6 MB
avg			-17.7%	-37.3%	-6.4%	-58.7%	-14.7%	-26.9%

Table 5.12: Solution times for our methods LP-2, A-2 and A-2+D on a hexalinear and a pseudo-orthoradial base graph. The relative approximation error of our A-2 approach is given as δ. If topology violoations occured, we do not give the approximation error, but the number of violations in parentheses.

	Hexalinear				Pseudo-Orthoradial			
	LP-2	A-2	δ	A-2+D	LP-2	A-2	δ	A-2+D
FR	2m	0.1s	10.1%	0.6s	41.7s	88ms	20.9%	0.2s
VN	8m	0.2s	6.7%	0.7s	30m	0.1s	— (1)	0.1s
CG	18h	1.7s	5.1%	3.1s	3h	0.4s	— (1)	0.8s
SD	30m	1.2s	7.4%	3.0s	13m	0.5s	5.3%	1.0s
BE	3h	1.3s	11.1%	3.3s	15h	0.8s	— (2)	1.1s
ST	23m	1.2s	8.5%	3.0s	9m	0.7s	— (2)	0.7s
LO	—	7.1s	— (10)	22.5s	—	12.4s	— (196)	9.1s
NY	—	30.4s	— (40)	2m	—	26.0s	— (180)	23.2s

5.11.5 Non-Octilinear Layout Experiments

Table 5.12 gives the running times and approximation errors of our A-2 and A-2+D approaches on a hexalinear and a pseudo-orthoradial grid (again, we could not evaluate the approximation error of A-2+D). Although the solution times are significantly higher than for the octilinear setting, our A-2+D approach was still able to produce orthoradial drawings for all but the London and New York datasets in under 1.1 seconds.

5.11.6 Visual Evaluation

Measuring the esthetic quality and the readability of schematic public transit maps (for example, through a user study) is a research topic of its own and would have been beyond the scope of this work. A recent survey by Wu et al. gives a thorough overview of the challenges [151]. To still give proof that the maps rendered using our approach are of high visual quality, various rendered maps of our testing datasets schematized by our approaches, including renderings which consider obstacles and approximate the geographical line courses, can be found at the end of this chapter. An additional visual comparison between our approach and other approaches from the literature can be found in Figure 5.4. All our schematic line graphs can also be inspected online[4].

[4] `https://octi.informatik.uni-freiburg.de/`

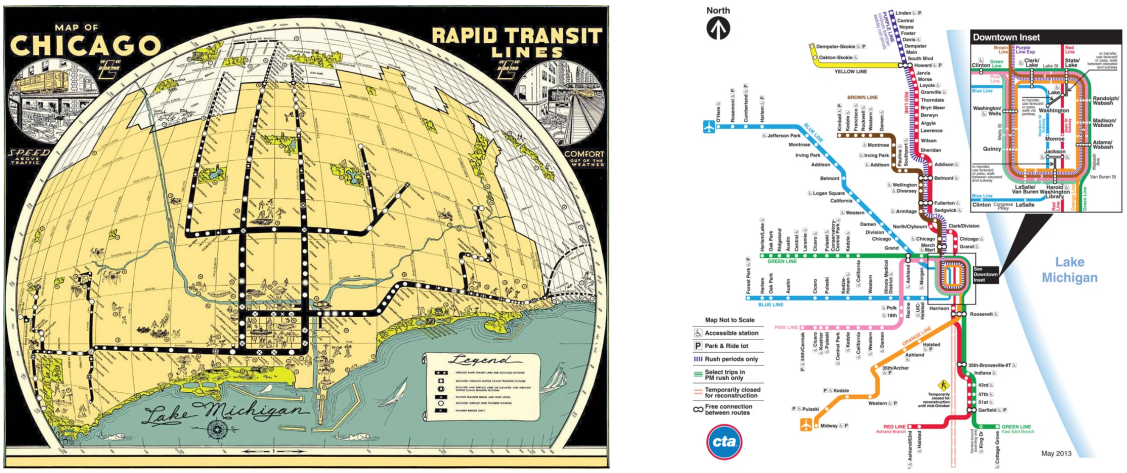


Figure 5.30: Left: A historical map of the Chicago light rail system in which the high-density Chicago Loop is enlarged using a fish-eye effect. Right: An official map of the Chicago network in which the Loop is depicted as an enlarged cut-out.

5.12 Conclusions and Future Work

This chapter described a novel approach to generate schematic drawings of transit maps by finding metro-map *images* of an input line graph in a special grid graph, with the ability to consider obstacles or approximate the original geographical line courses. Motivated by an NP-hardness proof for the general problem on octilinear grid graphs, we presented an ILP to find such images. As this ILP proved to be impractically large and slow to solve, we developed an approximation method that routes the input edges through the grid graph in a predefined order. To avoid stalling of this approach, we applied several smart heuristics and relaxed hard constraints for topology preservation. To further refine the metro-map images and to fix hard constraint violations in the initial drawing, we added a local-search step.

We demonstrated that our approach is able to find high-quality octilinear metro-map drawings fast (typically in under 1 second for small to intermediate-size networks) and that it is also able to draw metro-maps following an ortholinear, hexalinear, or (pseudo-)orthoradial layout. In an attempt to further speed up both the ILP solution and our approximate approach, and to reduce the memory consumption, we presented and evaluated several ideas for *sparse* base grids. We have also shown that the ability of our approach to consider obstacles or approximate geographical line courses is not just theoretical. Nevertheless, we regard our work in this chapter only as a first step towards fully automated schematic transit map generation using our approach. We do not yet consider the space needed for line or station labels during our schematization process. A posteriori labeling is possible, but will either fail if the schematization process did

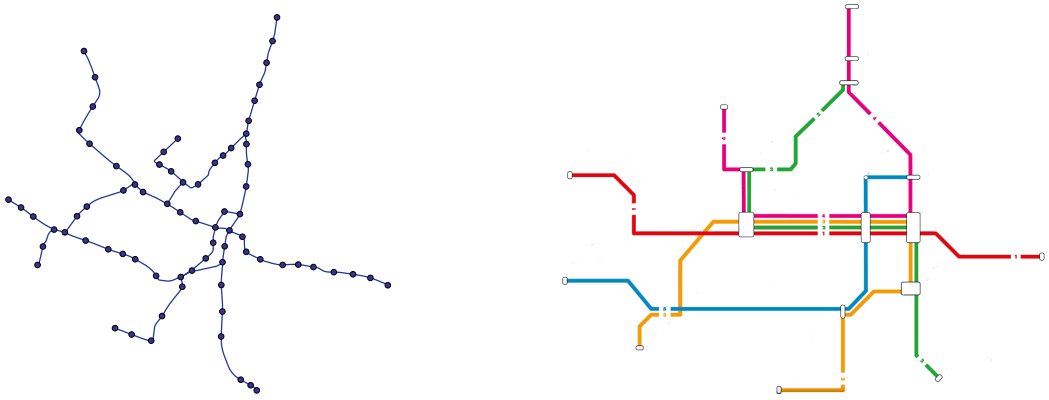


Figure 5.31: Left: Line graph for the 2020 Freiburg tram network. Right: A (simplified) official schematic representation of the network. The network has been rotated by around 40° in counter-clockwise direction to improve the alignment of the main axes.

not leave enough space for a label, or may not allow for an optimal placement. Future research has to address this problem.

We are also confident that more elaborate heuristics to prevent stalling of our approximate approach would help to reduce the number of topology violations for very large networks. Also, we so far only used the constraint relaxation for our heuristic approach. It may also help our ILP to arrive at a feasible solution faster, and may enable us to use a constraint-relaxed solution by our heuristic approach as a warm start for the ILP solver. Besides this, we see the following directions for future work:

Stott et al. [133] already noted that when octilinear metro-map embeddings are generated by a local search over the station positions on a regular grid it may be beneficial to try not only moving individual nodes, but entire *clusters* of nodes to overcome a local optimum. This idea directly translates to our approach, and using their clustered local search may further improve our approximate maps.

Our approach so far respects the positions of the original input stations by punishing station displacement. However, it is often desirable to enlarge certain parts of the network in a metro-map drawing, for example to ensure the readability of high-density areas by enlarging them into areas of lower density (for example, a city center enlarged into the municipal area). Such transformations are common in real-world maps, where the city-center is sometimes even depicted as an enlarged cut-out, or focused using a fish-eye effect (see the examples of the Chicago loop in Figure 5.30). The latter technique heavily distorts the map and violates octilinearity, while the former makes it hard to follow lines which cross the cut-out boundaries.

We see several ways to address this problem, they all transform the input line graph prior to schematization: one idea is to use a quadtree to identify areas of high density, which are then resized into empty neighboring cells. This transformation is completely

local and would not affect other parts of the network. A second idea is to identify station clusters in the input line graph using for example the k-means algorithm, and then applying a focus transformation on the center of each cluster with a density above a certain threshold. Last, we may use a classic spring embedder on the input line graph to arrive at a uniform distance between adjacent stations.

Real-world input line graphs also often feature one or more major *corridors* which are often drawn as straight vertical or horizontal segments in manually designed maps, even though their course is not strictly following an east/west or north/south direction. In effect, the manually designed maps are slightly rotated to enhance their appearance (see Figure 5.31 for an example). We see two simple ways to do this automatically: either we search for the rotation of the input graph for which the cumulated deviation of the segments from the east/west or north/south axes is minimized (for example by checking each rotation within a tolerable limit from $+45°$ to $-45°$ in 5 degree steps), or we octilinearize all of these rotated line graphs explicitly and pick the metro-map image of best score.

In the remainder of this chapter, we show several example maps rendered by our approach.

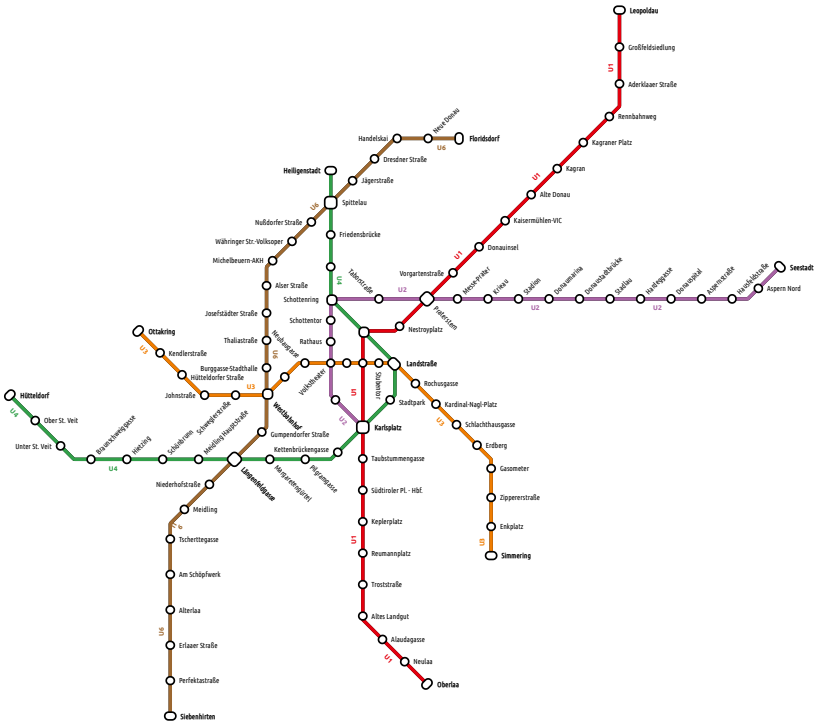


Figure 5.32: Octilinear map of the 2020 Vienna subway network rendered by our approach.

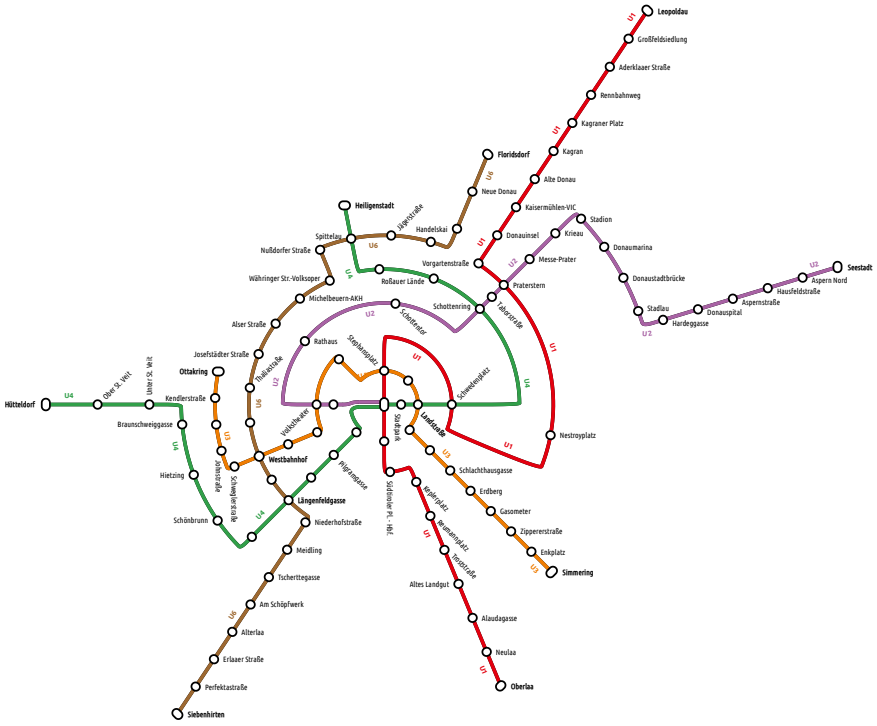


Figure 5.33: Orthoradial map of the 2020 Vienna subway network rendered by our approach.

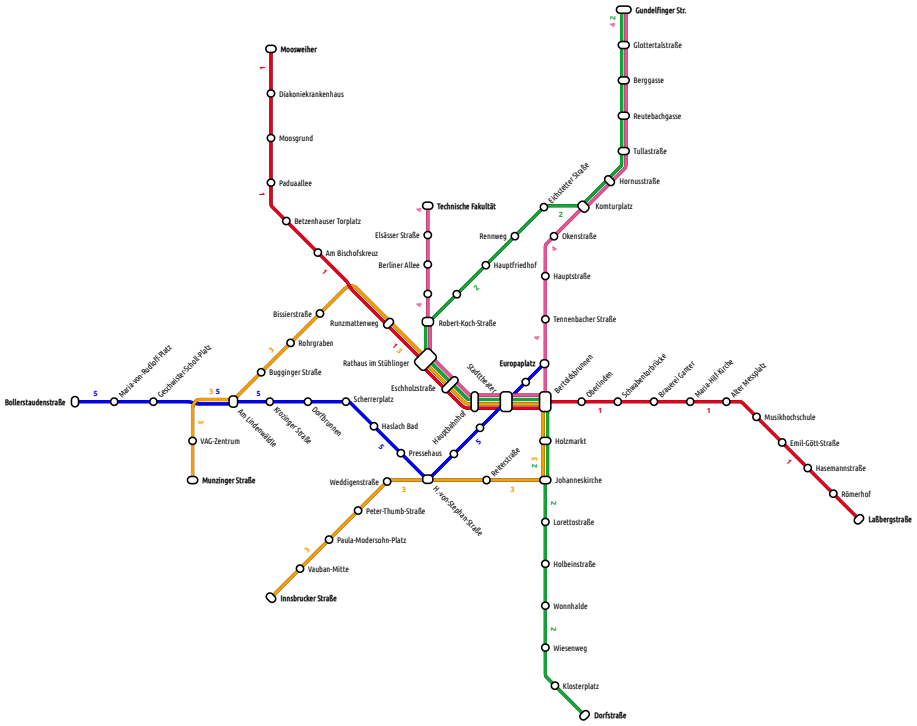


Figure 5.34: Octilinear map of the 2020 Freiburg tram network rendered by our approach.

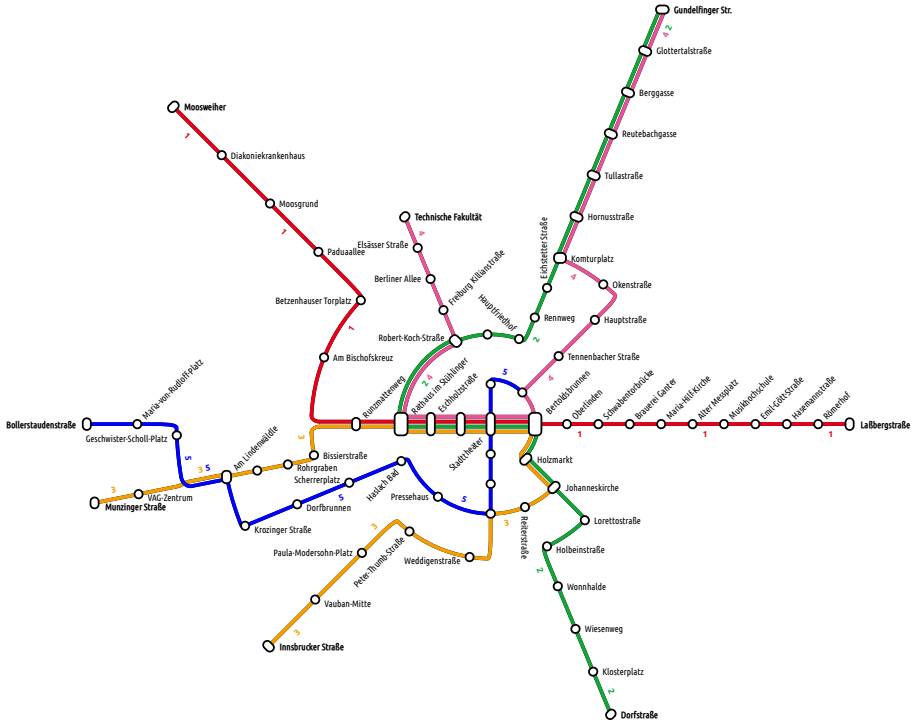


Figure 5.35: Orthoradial map of the 2020 Freiburg tram network rendered by our approach.

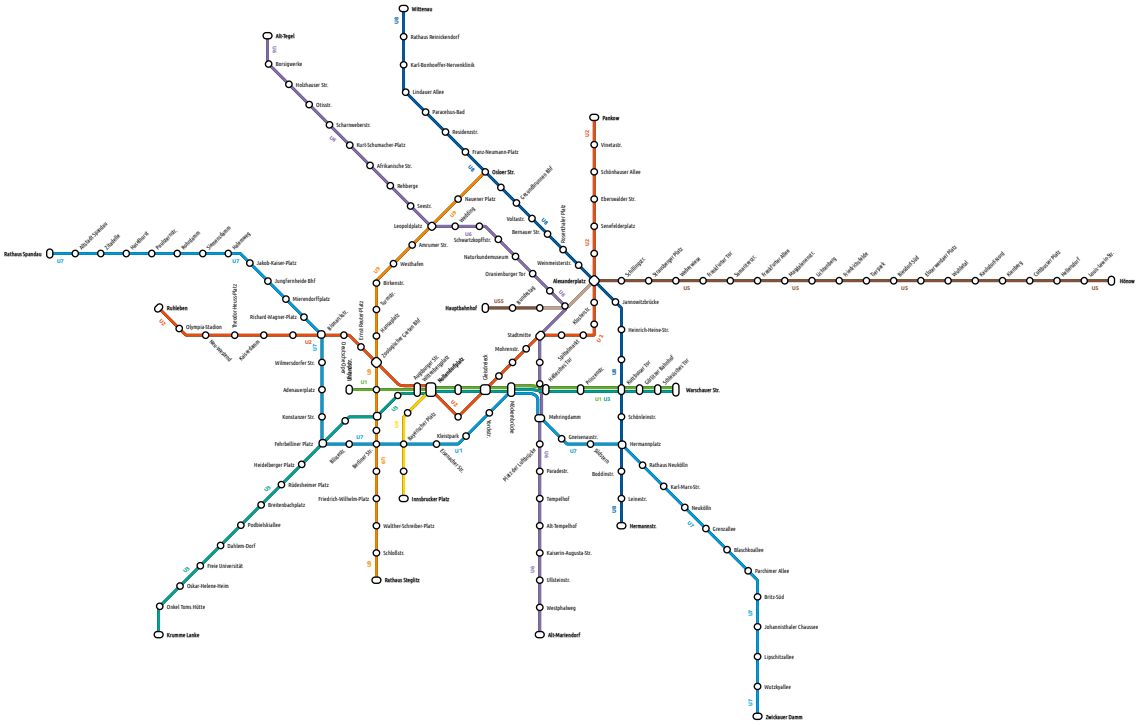


Figure 5.36: Octilinear map of the 2020 Berlin subway network rendered by our approach.

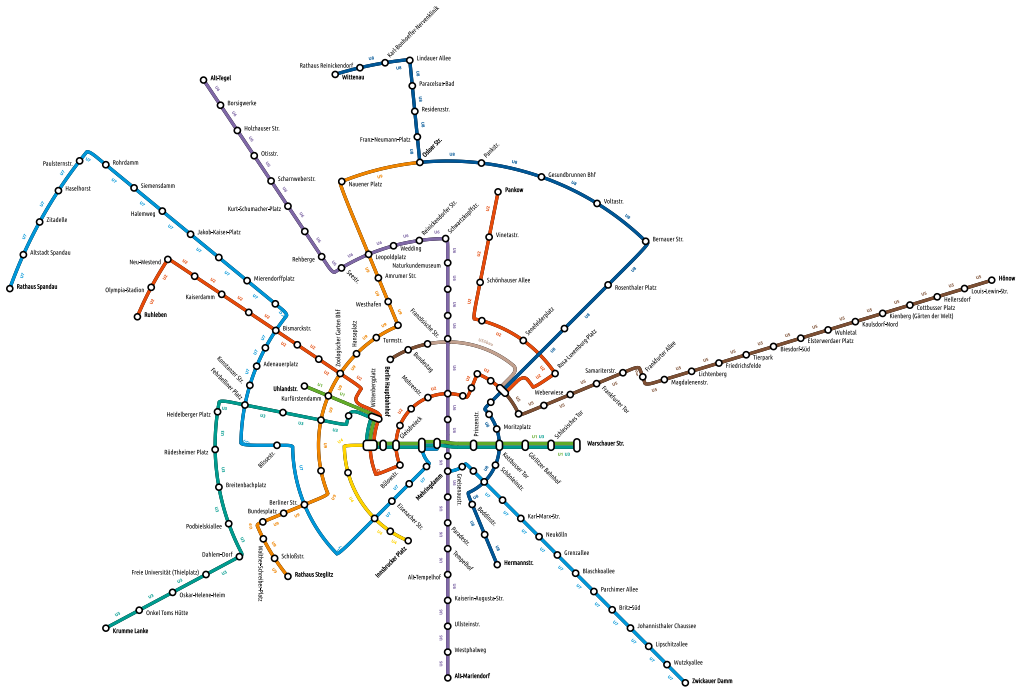


Figure 5.37: Orthoradial map of the 2020 Berlin subway network rendered by our approach.

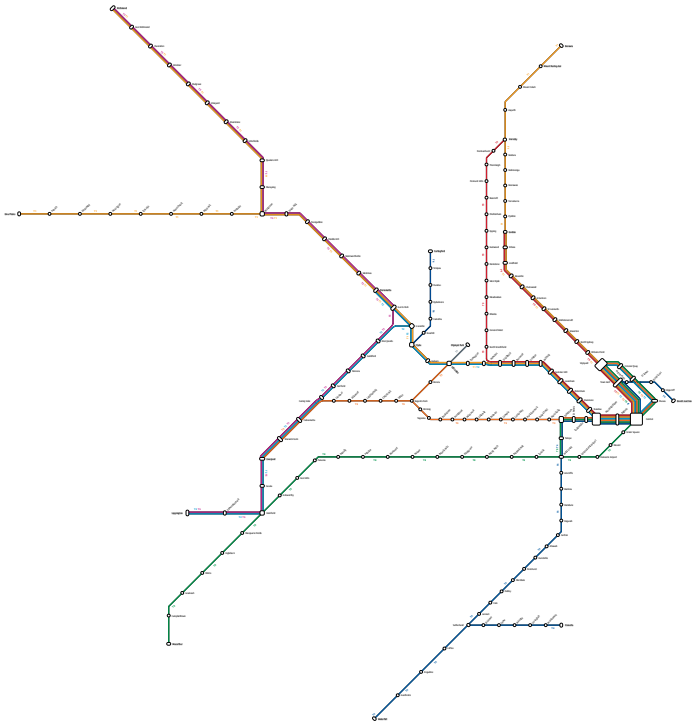


Figure 5.38: Octilinear map of the 2020 Sydney light rail network rendered by our approach.

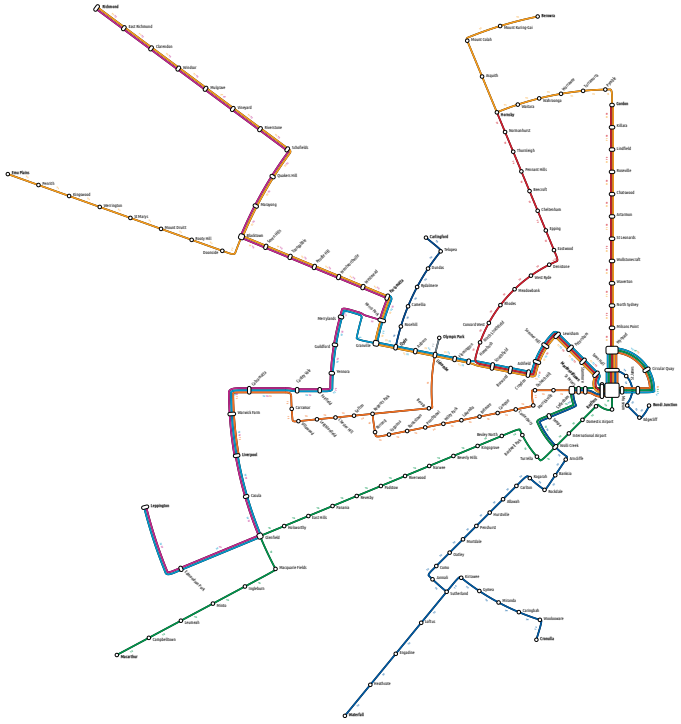


Figure 5.39: Orthoradial map of the 2020 Sydney light rail network rendered by our approach.

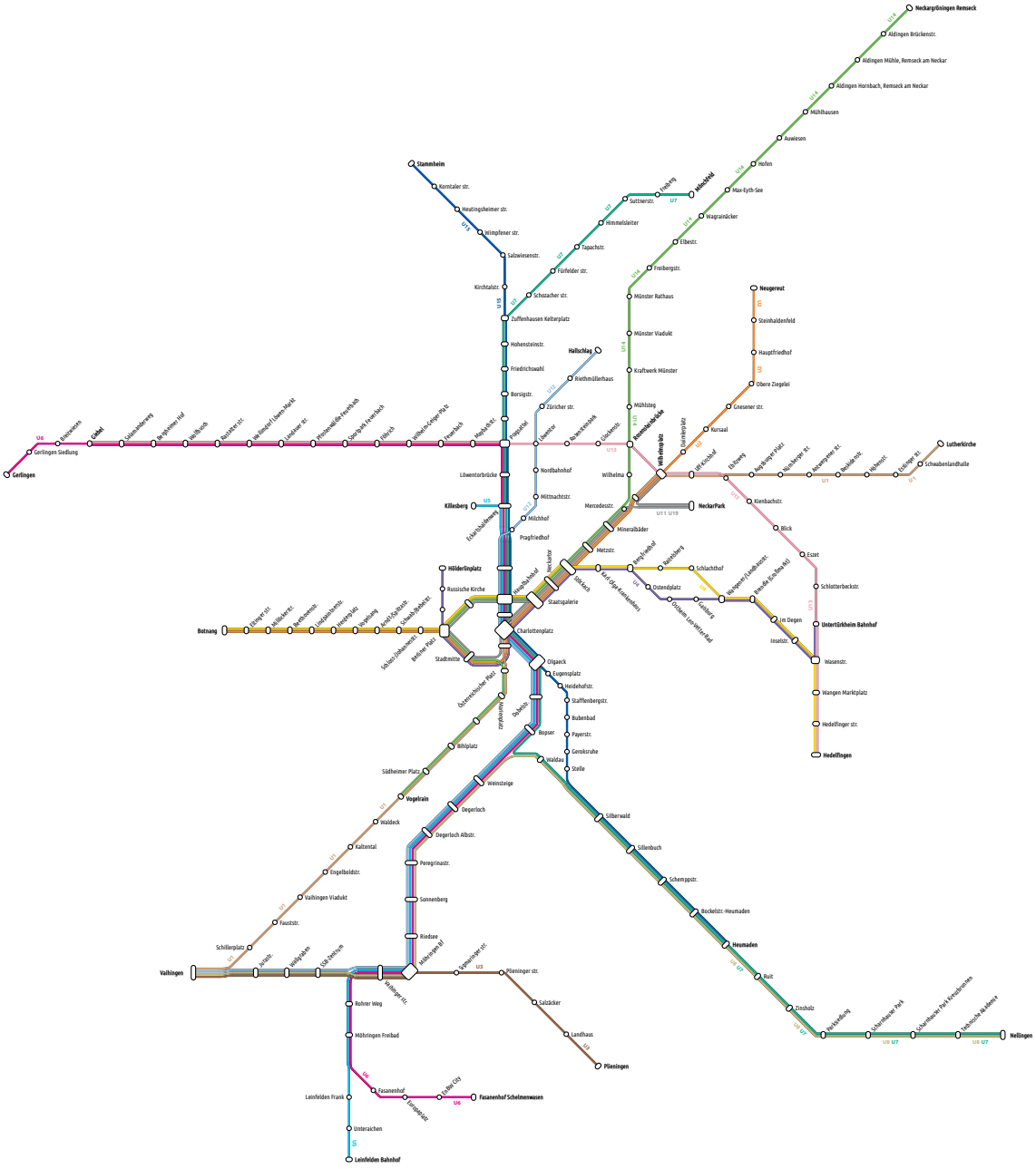


Figure 5.40: Octilinear map of the 2015 Stuttgart light rail network rendered by our approach.

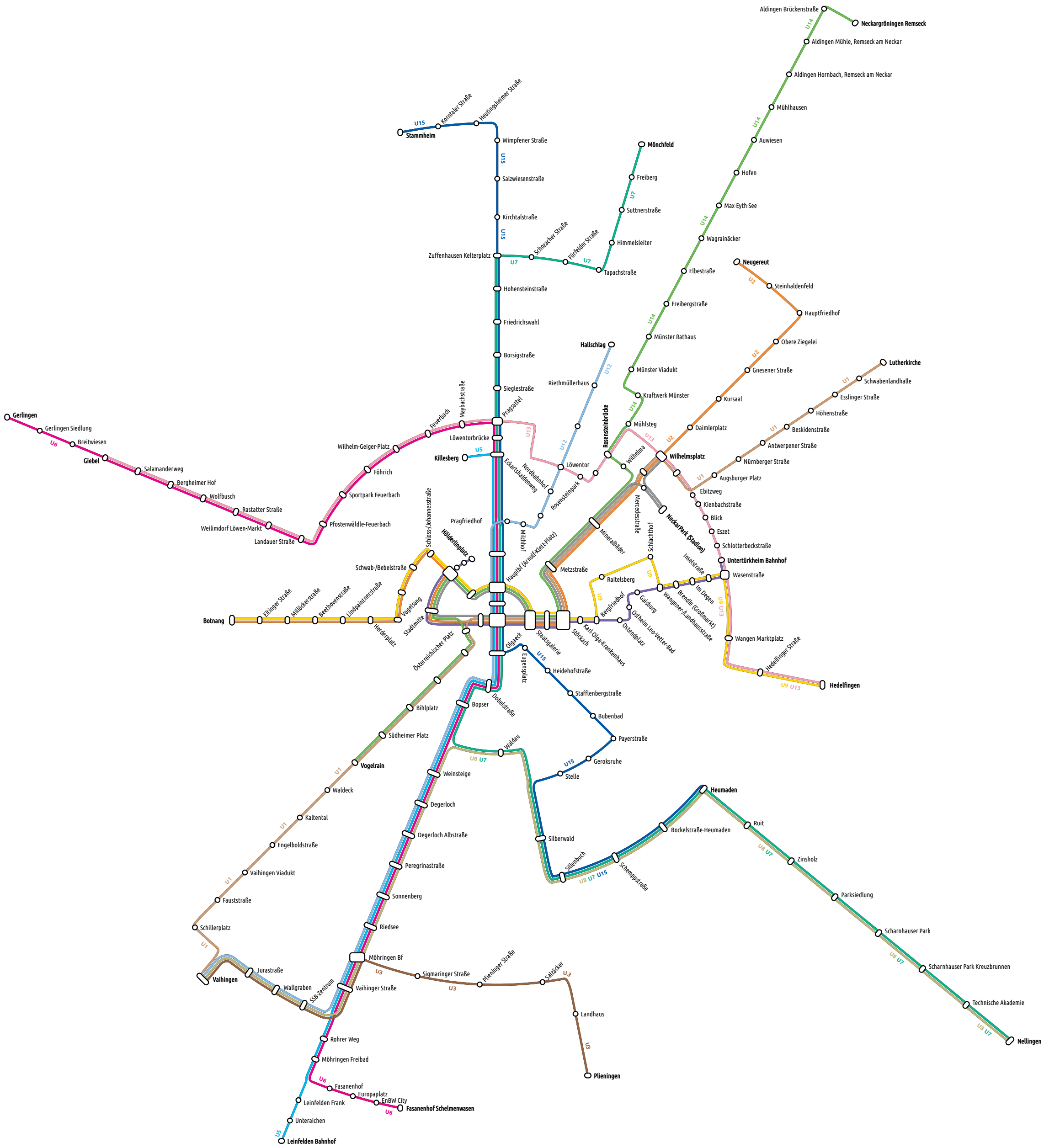


Figure 5.41: Orthoradial map of the 2015 Stuttgart light rail network rendered by our approach.

Figure 5.42: Hexalinear map of the 2020 Vienna subway network rendered by our approach.

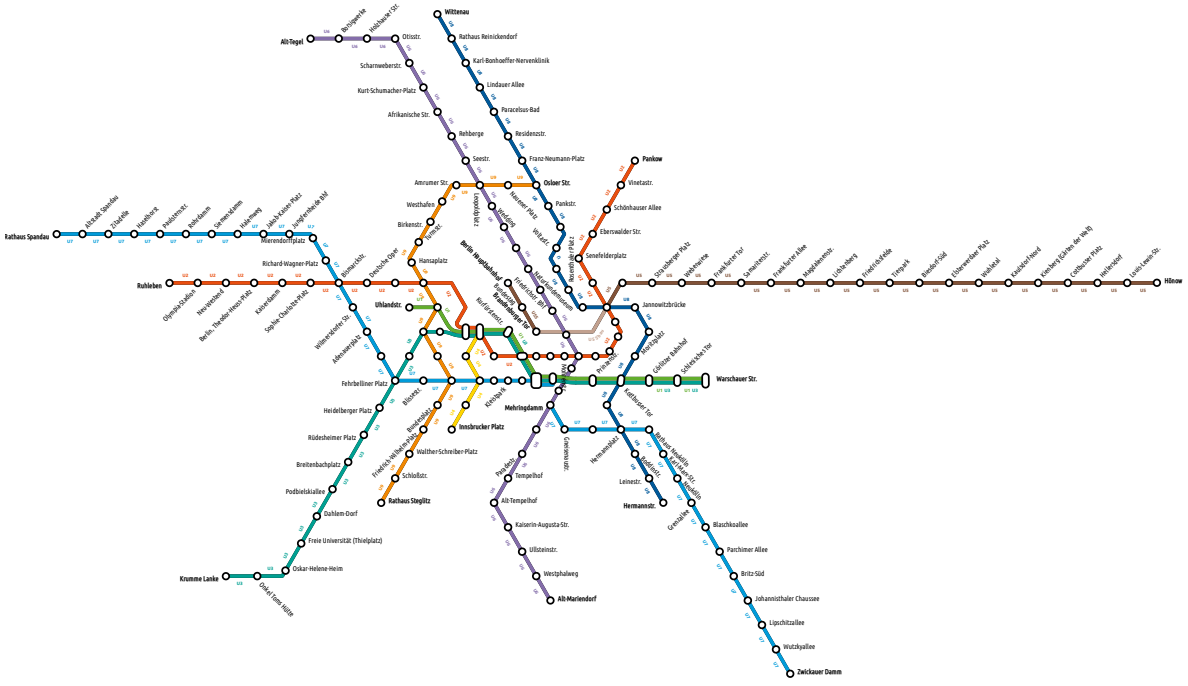


Figure 5.43: Hexalinear map of the 2020 Berlin subway network rendered by our approach.

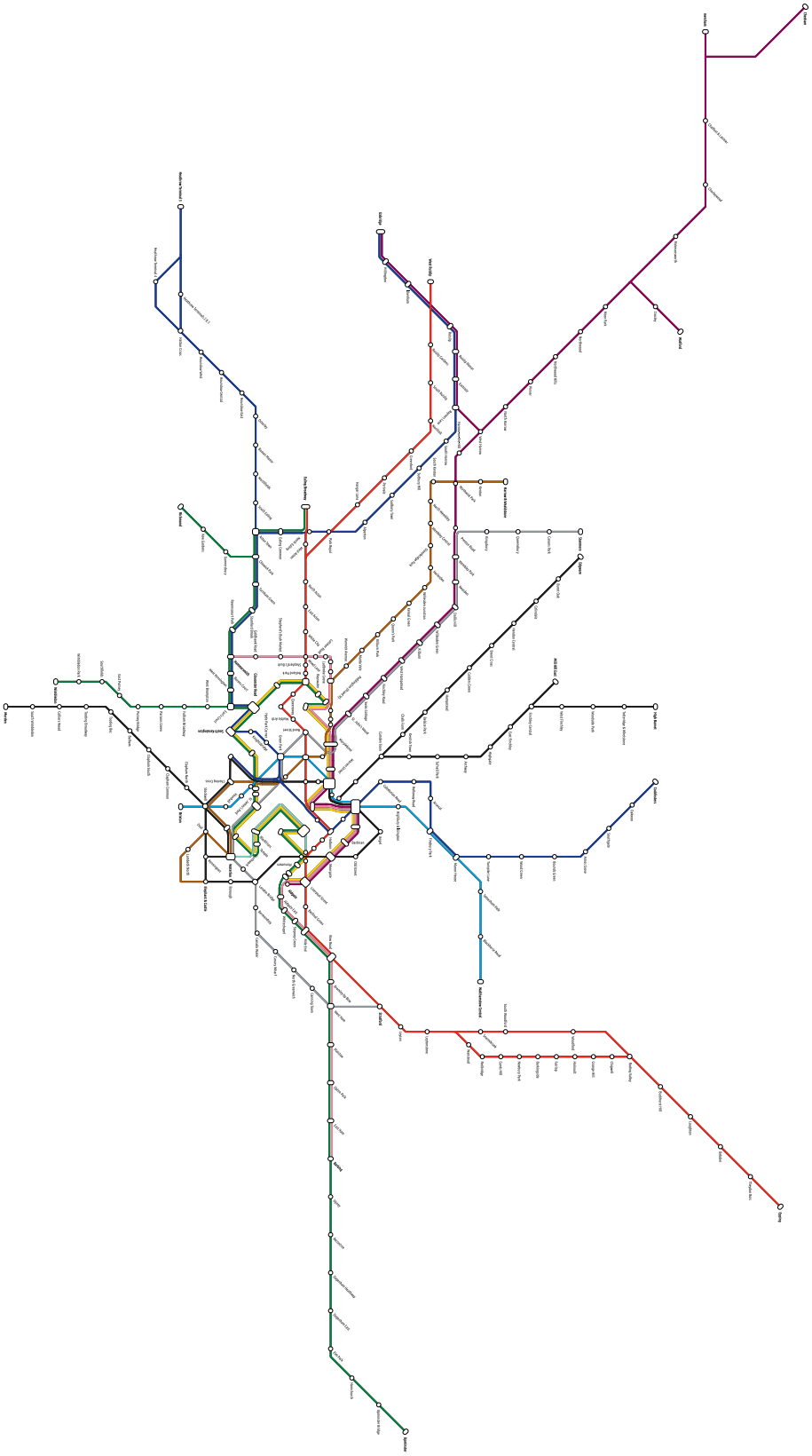


Figure 5.44: Octilinear map of the 2020 London subway network rendered by our approach.

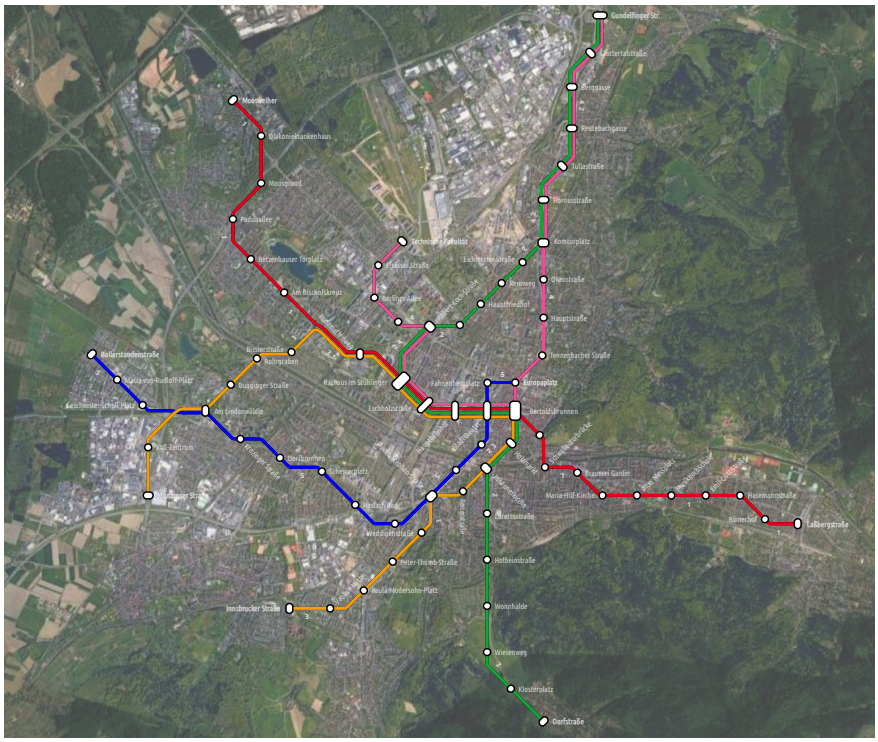


Figure 5.45: Octilinear map of the 2020 Freiburg tram network approximating geographical line courses, overlayed over satellite imagery.

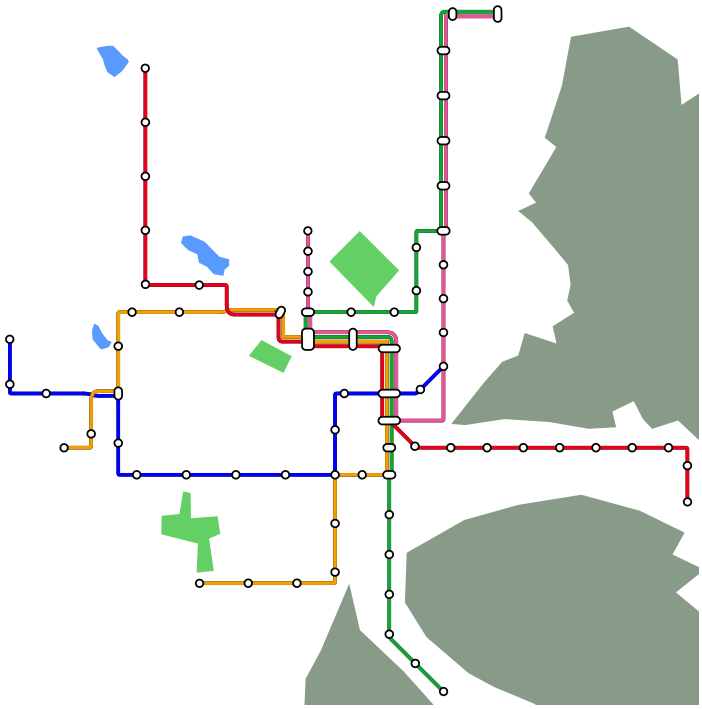


Figure 5.46: Ortholinear map of the 2020 Freiburg tram network, respecting obstacles of parks, forest, and mountains.

Chapter 6

Rendering and Labeling

We close the algorithmic part of this work with a short and practical description of how the line graphs generated by the methods presented so far can be rendered into a map. This chapter thus addresses the following problem:

Problem 6 (Line Graph Rendering)**.** *Given a line graph $G = (V, E, L, \mathcal{L})$ and a line ordering for G, render G into a map.*

The input line graph G may be either geographically correct, or schematic. We assume that the line graphs have been prepared in a way resembling the approach described in Chapter 3. In particular, this means that edge segments within a threshold distance $\hat{d}$ have been merged.

6.1 Overview

As mentioned in the introduction of this work, we chose our definitions of a line graph and a line ordering in such a way that they will allow for easy rendering. Our basic approach, depicted in Figure 6.1, is hence straightforward: Given an input line graph G (1), we first render all lines $L(e)$ passing through an edge e in the order given by the respective line ordering as offset polylines (2). Recall that although G is undirected, we always give the line orderings w.r.t. to an adjacent node to ensure that the rendering order is well-defined. Observe that adjacent line segments will inevitably overlap at intersection nodes. In a process we call *node expansion*, we therefore first free the intersection nodes by these overlapping parts (3) and reconstruct the intersection by connecting lines in adjacent edge segments with Bézier curves (4). Then the stations are rendered on top (5). In a final and optional step, labels are added for the lines and the stations (6). Each of these steps will be further explained below.

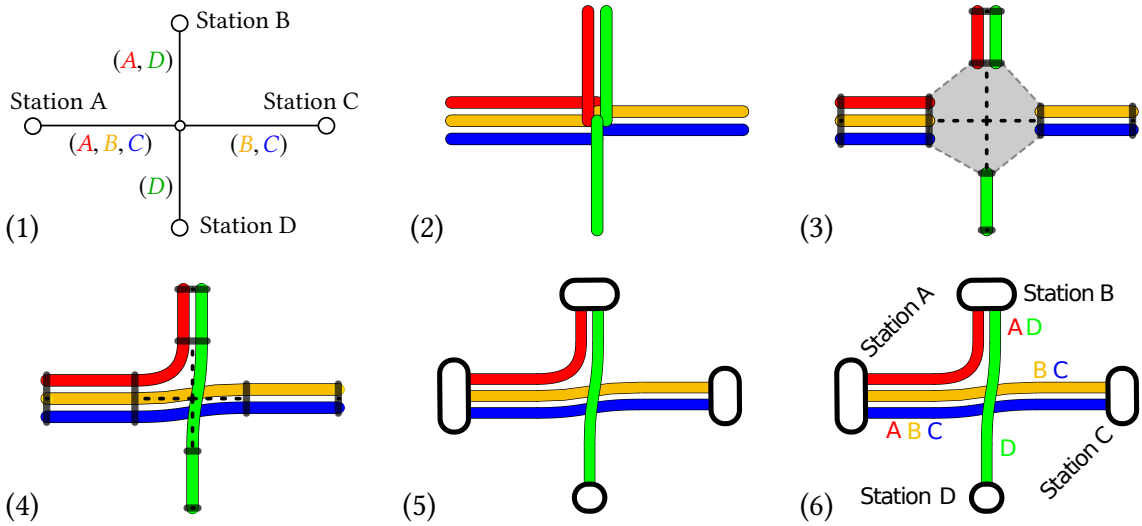


Figure 6.1: Overview over our rendering approach. Given an input line graph (1), render the ordered lines as offset polylines (2), expand nodes (3), reconstruct node connections with Bézier curves (4), render station markers (5), and optionally label the map (6).

6.2 Offsetted Line Rendering

Let e be an input edge $e = \{u, v\}$ which is embedded in $\mathbb{R}^2$ be a polyline $P(e)$, and let $(l_1, \cdots, l_n), l_i \in L(e)$ be the line ordering for e. W.l.o.g., we assume that the line ordering is given w.r.t. to v. We then render each l_i as a polyline of width w which is perpendicularly offsetted from the supporting polyline $P(e)$ by $-w\,|L(e)|\,/2+w\,(i-1)$. If $P(e)$ is a straight line segment, this is simple. Offsetting an arbitrary polyline, however, is nontrivial, and known as (open) polygon offsetting, or polygon buffering, a process which is closely related to the problem of constructing a *straight skeleton* of a polygon [4]. A straight skeleton of a polygon P can be thought as being composed of lines tracking the nodes of P when P is shrunk.

As polygon buffering is a common requirement in many fields, many approaches have been proposed in the literature (see [34] for an overview). The general difficulty is that when the polygon sides are offsetted individually in a perpendicular fashion, they will often overlap and introduce self-intersections.

For this work, we used a relatively simple approach for offsetting the supporting polyline, as depicted in Figure 6.2: let $P = (s_1, \cdots, s_n)$ be an open polygon, defined by its straight line segments $s_i = (p_i, p_{i+1})$, that should be offsetted by some distance d. We iterate over the segments in the order given by P and perpendicularly offset each s_i by d, resulting in a new offsetted segment s'_i. The lines (not the line segments!) described by s'_i and the previously inserted segment s'_{i-1} intersect at some point q, which we add as a new support point for P' (for the first end last segment, we simply keep the first (or last)

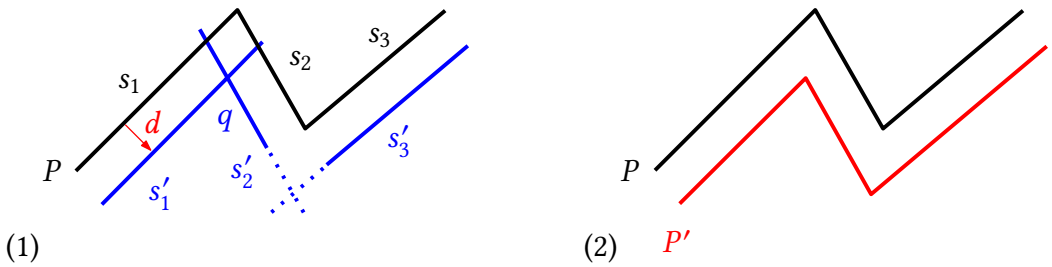


Figure 6.2: Offsetting a polyline (an open polygon) by a distance d. (1) We iterate over the polyline segments $s_1, \ldots, s_n$, offset each s_i by d and determine the intersection q of the lines described by the offsetted segment s'_i and the last offsetted segment s'_{i-1}. q is added as a new support point to the offsetted polygon P' (2).

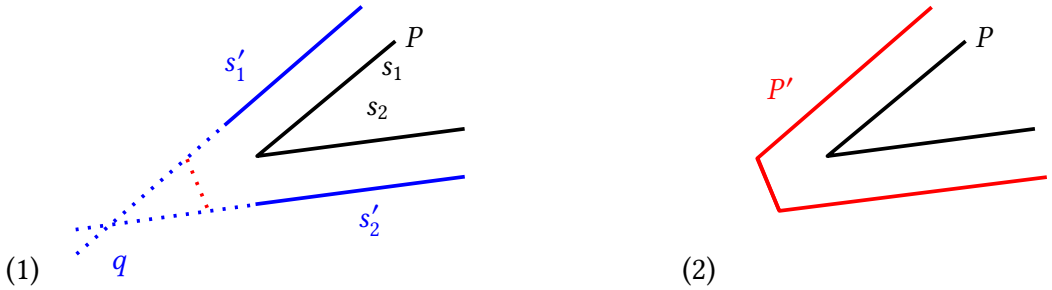


Figure 6.3: (1) If the intersection point q of the lines described by two offsetted line segments s'_1 and s'_2 is too far away from s'_1 or s'_2, we cap the resulting spike as depicted. (2) The resulting offsetted polyline.

point of the offsetted segment). Note that if s'_i and s'_{i-1} are colinear, the intersection point q is also not well-defined. We avoid such cases by first simplifying the input polyline using the Douglas-Peucker algorithm.

If two adjacent segments s_1 and s_2 meet at an acute angle, the line intersection point q of their offsetted segments s'_1 and s'_2 may be far away from s'_1 and s'_2, as shown in Figure 6.3. This produces undesirable spikes in the final map rendering. To mitigate this effect, we clip the corresponding segments if the distance is above the offset distance d and introduce a new connecting segment, as shown in Figure 6.3.2.

The approach described so far only considers local intersections between adjacent segments, and topological errors may still be introduced by segments which do not appear consecutively. Figure 6.4 gives an example. We clean such errors in a post-processing step in which we detect self-intersections and prune the corresponding loop from P (Figure 6.4.3).

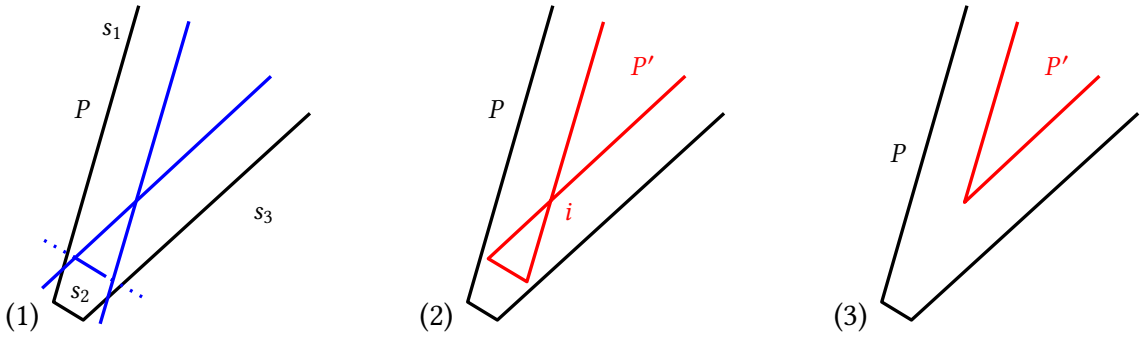


Figure 6.4: (1) Our offsetting approach may fail to catch intersections between offsetted line segments which were not originally adjacent in P. This will results in topologically incorrect self intersections in the offsetted polyline (2). We apply a filtering step in which we detect such self-intersections and delete the corresponding loop (3).

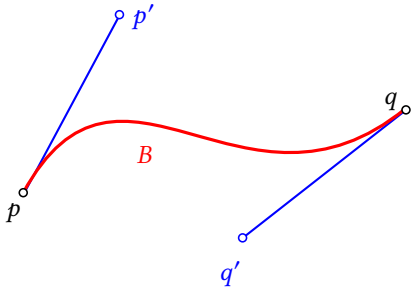


Figure 6.5: A cubic Bézier curve B with its four control points p, p', q', and q.

6.3 Inner Node Line Connections

The packed offsetted lines of each edge s$\{u, v\}$ can now be thought of describing a polygon. We call the polygon segments that touch u or v the *node fronts* of e for the respective nodes, and denote the node front of e at u as F_{eu}. To make room for the inner node connections at intersection nodes, we now move for each node v all adjacent node fronts simultaneously inward with fixed step sizes until a stopping criterion is met (we use a distance threshold between the nodes). Figure 6.1.3 gives an example. As mentioned above, we then connect corresponding lines with a Bézier curve. For our work, we used cubic Bézier curves, as defined in the following [58].

Definition 6.1 (Cubic Bézier Curve)**.** Given four *control points* p, p', q, and q' in $\mathbb{R}^2$, a cubic Bézier curve is defined by a polynomial function $B : [0, 1] \mapsto \mathbb{R}^2$:

$$B(t) = (1-t)^3 p + 3(1-t)^2 t p' + 3(1-t)t^2 q' + t^3 q. \tag{6.1}$$

p is the starting point, q the endpoint of B. Intuitively, $p - p'$ and $q - q'$ can be thought of defining vectors which describe the direction of the curve at p and q, and the magnitudes $|p - p'|$ and $|q - q'|$ describe how much the curve is "forced" to follow that

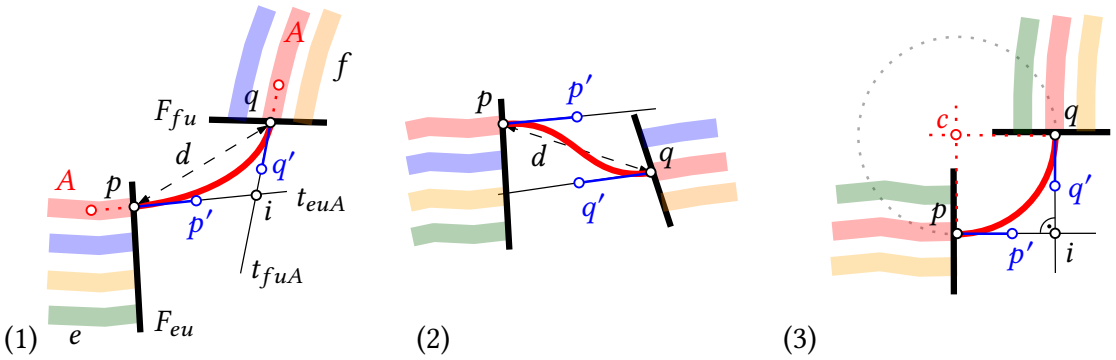


Figure 6.6: (1) Inner node line connections for a line A between two node fronts F_{eu} and F_{fu} drawn as cubic Bézier curves between p and q with control points p' and q' positioned on line segments t_{euA} and t_{fuA} (both of length $d = |p - q|$) following the direction of A on the last 5 meters before reaching the node point geometries. The position of p' and q' on t_{euA} and t_{fuA} is based on the lengths $|p - i|$ and $|q - i|$. (2) If t_{euA} and t_{fuA} do not intersect, the position is based directly on d. (3) Bézier curve approximating a circular segment with center c.

direction. B is tangential to the lines pp' and qq'. Figure 6.5 gives an example.

Let now A be a transit line passing through a node u from adjacent edge e to adjacent edge f, p the position where A arrives at the node front F_{eu}, and q the position where A arrives at the node front F_{fu} (Figure 6.6.1). We first determine unit length vectors d_{euA} and d_{fuA} which describe the direction of transit line A at the respective node front by sampling the corresponding polylines of A close to F_{eu} and F_{fu} (for our experiments, we used a distance to the node fronts of 5 meters). Using these direction vectors, we then draw two line segments t_{fuA} and t_{euA} starting at p and q, respectively, and going into the direction d_{euA} and d_{fuA}. As a length of t_{fuA} and t_{euA} we use the distance $d = |p - q|$.

There are now two cases: if t_{fuA} and t_{euA} intersect at some point i, we average the length of the segments pi and qi, that is we compute a $\delta = (|p - i| + |q - i|)/2$. The control point p' is then placed at $p + d_{euA} \cdot k \cdot \delta$, and the control point q' is placed at $q + d_{fuA} \cdot k \cdot \delta$ (Figure 6.6.1). We set $k = 4/3(\sqrt{(2)} - 1) \approx 0.55228474$. If t_{fuA} and t_{euA} do *not* intersect, they are parallel, or close to parallel, and we simply place the control points at $p' = p + d_{euA} \cdot k \cdot d$, and $q' = q + d_{fuA} \cdot k \cdot d$ (Figure 6.6.2).

A particularly nice property of this setup using the magic number $k = 0.55228474$ is the following: if the intersection point i exists, and if $|p - i| = |q - i|$, then $\delta = |p - i| = |q - i|$, and the corresponding cubic Bézier Curve B approximates a circular arc connecting p and q [125] (the approximation error is negligible in practice [73]). This makes for a particularly clean look of schematic networks. Figure 6.6.3 gives an example for the orthogonal case where the circle center is at $c = q - \delta d_{euA}$. However, it is easy to see that d_{euA} and d_{fuA} do not have to be orthogonal.

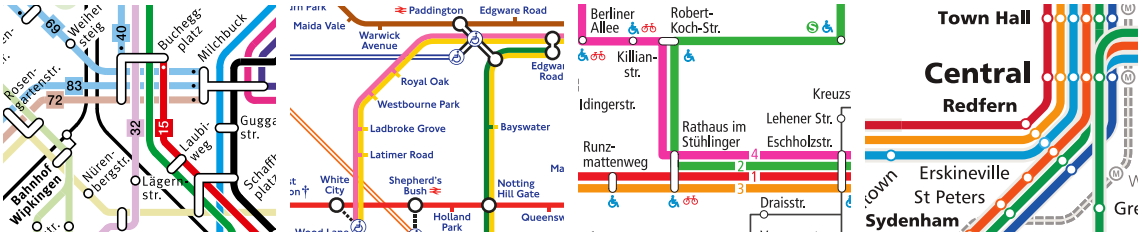


Figure 6.7: Different station marker styles in (from left to right) the official transit maps of Zurich, London, Freiburg, and Sydney. For another styling example, also compare Figure 4.41.

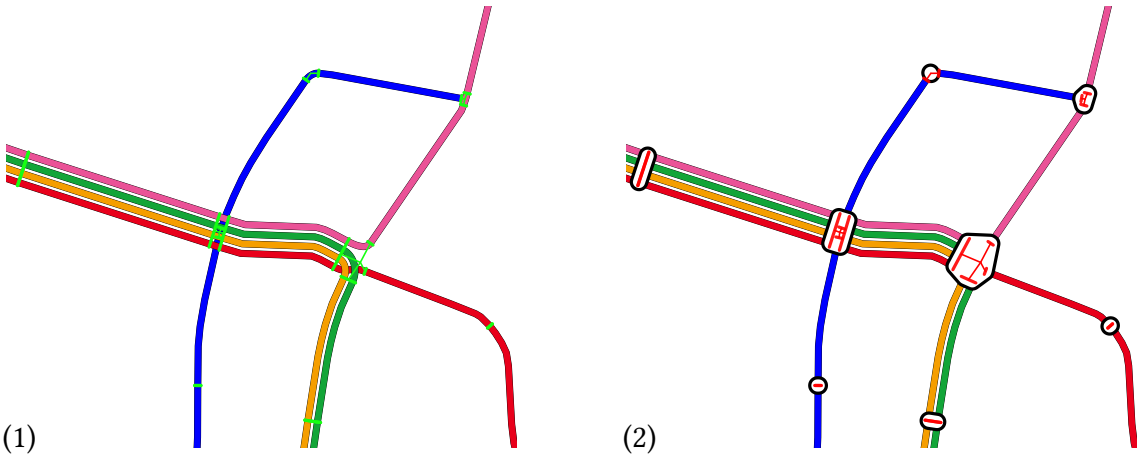


Figure 6.8: (1) Excerpt from a rendered map of the Freiburg tram network, without station markers. Expanded node fronts are shown in green. (2) Stations rendered by taking the convex hull of the node front geometries, with node fronts overlayed in red.

6.4 Station Rendering

For the station rendering, we have to consider two cases: either a station is served by *all* adjacent lines, or some lines pass through u without a stop.

6.4.1 Fully Served Stations

Let $u \in S$ be a station node with corresponding node fronts $F_1, \cdots, F_n$. To render a station which is served by all adjacent lines, we simply take the padded convex hull of $\{F_1, \cdots, F_n\}$ and use the resulting polygon as a station marker, as shown in Figure 6.8 (for stations of degree 2 that only serve a single line we use a simpler approach and render them as round markers).

While taking the convex hull produces compact station markers, real-world public transportation maps often feature rectangular station nodes. To achieve this look, we

Figure 6.9: (1) Station marker rendered using the convex hull of the node fronts (2) Station marker rendered using the oriented bounding box of the node fronts.

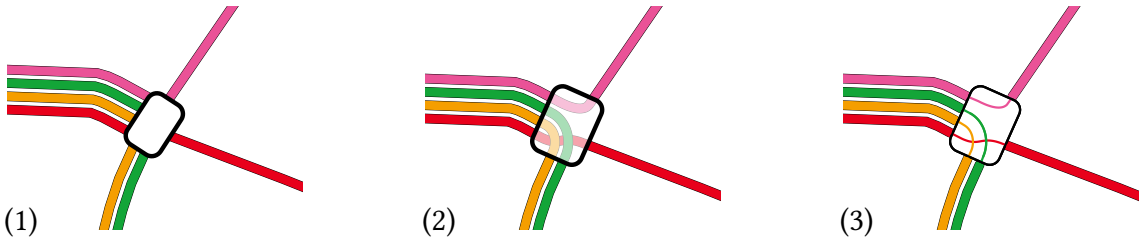


Figure 6.10: (1) Station marker using the oriented bounding box of the *unexpanded* node fronts, producing a tighter look. (2) First expanding the node fronts allows for a clean visualization of the line courses, for example by making the station node transparent.

instead compute the oriented bounding box of the $\{F_1, \cdots, F_n\}$. Figure 6.9 compares both approaches. Note that the node front expansion is not strictly necessary for station nodes if they are rendered in this fashion. In fact, leaving the node fronts unexpanded for station nodes will produce tighter station markers, as shown in Figure 6.10.1. However, also expanding station nodes allows to interactively hide the station markers, or to render stations in a way which makes it possible to follow the line courses through the stations. Figure 6.10.2 and 3 gives two examples. Also recall Figure 1.3, where stations were drawn as transparent polygons.

Figure 6.11: Station markers rendered directly on the node fronts for stations not served by all adjacent lines. (1) The pink line does not serve the center stop. (2) Both the pink line and the orange line do not serve the stop.

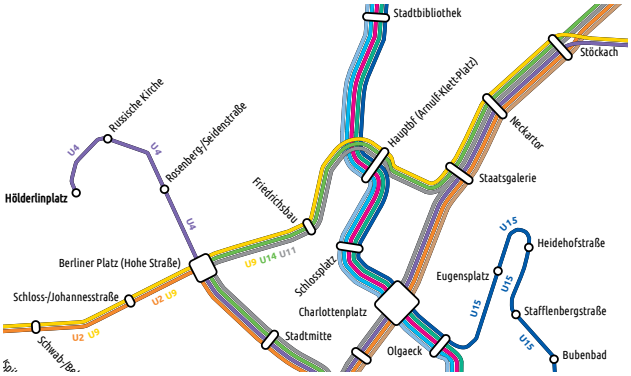


Figure 6.12: Excerpt from the Stuttgart light rail network map, labeled using our preliminary approach. Terminus labels are rendered in bold text.

6.4.2 Partially Served Stations

For stations that are only served by some adjacent lines, we take a simpler approach and render the station markers directly onto the node fronts. Figure 6.11 gives two examples. Let u be a station node adjacent to lines $L = \{l_1, \cdots, l_n\}$, and let $N \subset L$ be a set of lines which do not serve u. We then iterate over each node front and add a station marker onto the incoming position of an unmarked line $l_i \notin L$, and mark l_i as processed. If two station markers are next to each other on the same node front, we merge them. An obvious extension to this method is to optimize the distribution of the station markers in such a way that it creates as few markers as possible. For example, in Figure 6.11.1, the method may place the station marker for the green and orange line onto the lower edge, leaving a single station marker for the red line to be placed on the left or right edge.

This style of station markers may also be used on stations served by all lines - consider for example the official maps of London and Sydney shown in Figure 6.7.

6.5 Labeling

Map labeling (sometimes called automatic label placement) is a large research topic of its own and can be considered an optimization problem: place a set of labels (each for a geometry on the map) in such a way that they do not overlap (conflict), and that as many labels as possible are visible. For surveys or literature, see for example [136] or the map-labeling bibliography maintained by Alexander Wolff [150]. It can be considered a geometric maximum independent set problem, in which two labels are adjacent in the graph-theoretical sense if they overlap. The problem is NP-hard [61]. Typically, there is a set of position candidates for each label (for example, placed to the left, right, top, or bottom). A simple approach which has found its way into some web mapping libraries[1,2] is to use a local search, typically simulated annealing [39], to approximate an optimal

[1] `https://github.com/migurski/Dymo`
[2] `https://github.com/tinker10/D3-Labeler`

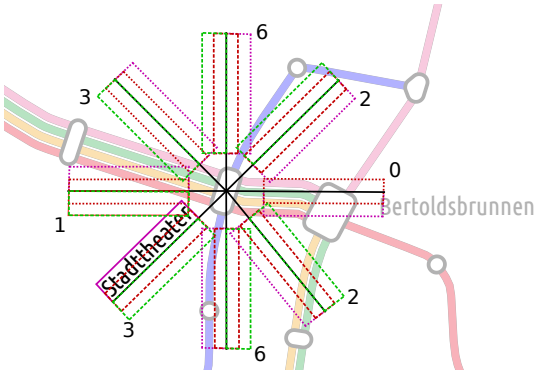


Figure 6.13: Placement candidates for a station label. At each octilinear orientation, weighted by a score, we place a candidate polygon (the bounding box of the text label) at 3 text baseline positions. The conflict-free candidate of lowest score is chosen.

labeling. The problem of labeling *dynamic* maps which change over time (for example by panning or zooming) or even feature moving objects has also found considerable research interest in recent years [72].

For transit maps, there are typically two classes of labels: station labels and line labels. For the latter, three approaches are predominant in manually designed maps: (1) the line labels are placed *on the lines* (see for example the Freiburg map in Figure 6.7), (2) the line labels are placed at their terminus stations, and (3) the line labels are placed in their respective color along the edge segments. In case (1), no collisions can occur. In case (2), the line label is part of the station label. We therefore only considered case (3) in our work so far. In the context of schematic transit maps, map labeling is often part of the schematization process, as the schematization may influence the possible label positions, and vice versa. As already hinted at in Chapter 5, map labeling was not the focus of this work. We nevertheless implemented a preliminary labeling method.

The approach we chose is a straightforward greedy approach which places the labels on the map consecutively: we first order the station nodes (in descending order) by their degree to ensure that larger stations are labeled first. For each label, we then rotate a polygon corresponding to the label's bounding box by multiples of 45° and check for each of these octilinear orientations whether a collision occurs (Figure 6.13). If so, we discard the position. If not, we assign each rotation a score, given here in clockwise direction starting at 0°: $(6, 2, 0, 3, 6, 4, 1, 5, 6)$. The scores favor positions in which the line labels are easy to read, and the reading order starts at the station node. We pick the position with the lowest score. For the line labels, we try positions along the corresponding edge and weight collision-free placements by their distance to the midpoint of the edge geometry. If no collision-free placement is available, we do not display the label. While the resulting labeling is certainly not perfect, we consider them satisfactory. Further work is required here, though. Figure 6.14 shows a labeled geographically accurate map of the Stuttgart light rail network, created using our approach described so far.

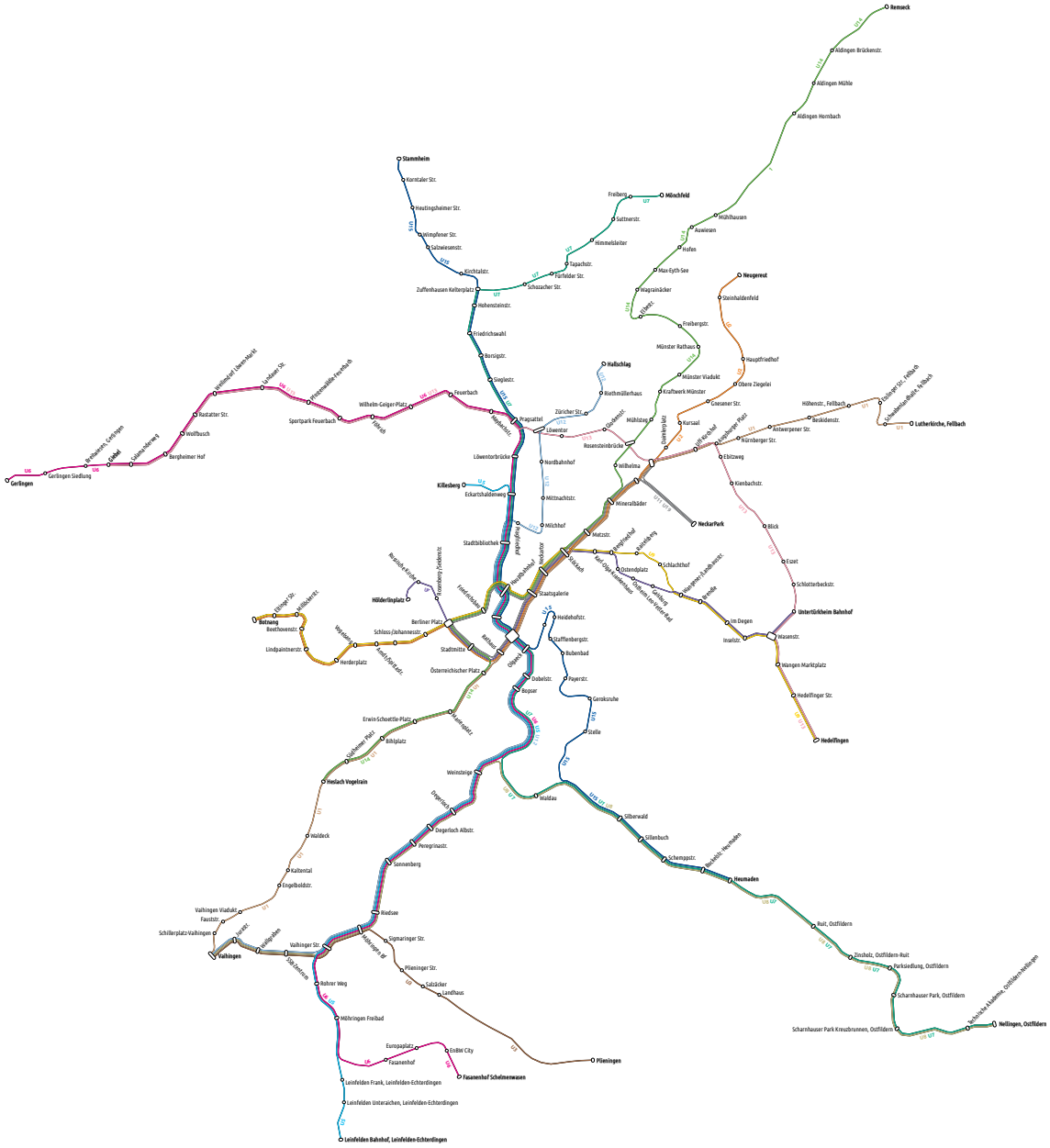


Figure 6.14: Labeled and geographically accurate map of the 2015 Stuttgart light rail network.

Chapter 7

Software

This chapter provides on overview over (and short manuals for) the software developed in the course of this work. The order in which the software is presented matches the order in which the implemented methods were presented in Chapters 2 to 6. We took the pipeline metaphor seriously: the last 4 tools presented here (*topo*, *loom*, *octi*, and *transitmap*) can be plugged together using Unix pipes, which makes it straightforward to extend our approach. All of our developed tools are open source and available online.

7.1 pfaedle - Map-Matching GTFS Data

The map-matching approach for public transit data presented in Chapter 2 has been implemented in a C++ tool called *pfaedle*[1]. Our tool expects an OpenStreetMap dataset and a schedule dataset given in the GTFS format as input, and outputs a fully map-matched GTFS dataset with a shape for each trip. For example, the following command will create shapes for a GTFS dataset in `freiburg-gtfs` from OSM data in `freiburg.osm` and write it to the (default) folder `gtfs-out`:

```
$ pfaedle -x freiburg.osm freiburg-gtfs
```

It is highly configurable. In particular, the process of building the network graph for different methods of transportation (MOT) can be controlled via extensive filter rules, and adding additional MOTs simply amounts to adding a new section in the configuration file. Other features include the ability to output filtered OSM files that contain exactly the part of the data needed to map-match one or more input GTFS feeds, or a matching Overpass API[2] query to obtain this data. Instead of outputting the shaped GTFS feeds, *pfaedle* may also output the network graph as a GeoJSON file with trips map-matched onto it.

[1] `https://github.com/ad-freiburg/pfaedle/`
[2] `https://wiki.openstreetmap.org/wiki/Overpass_API`

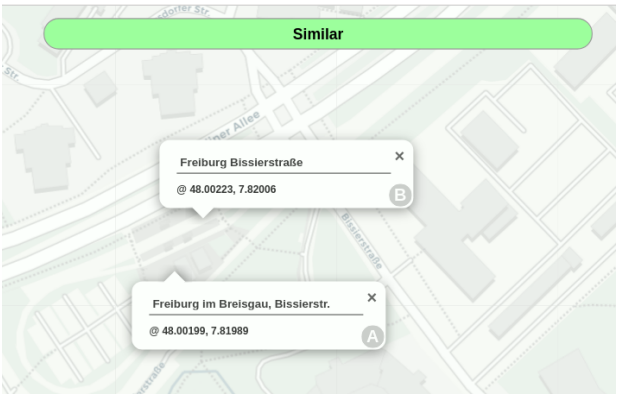


Figure 7.1: Station similarity classification GUI for *statsimi*. Users may click on a map to add station position markers, which can then be labeled freely. The classification result is displayed on top.

pfaedle was published in 2018. Based on feedback we received from various organizations and transit companies, it has since become a standard tool for the generation of missing shape information for GTFS feeds. It is also used in our transit visualization tool TRAVIC[3] [17, 18] to weekly map-match many of the GTFS input dataset (among them the nationwide schedule datasets for Germany, Switzerland, Sweden, and Denmark).

7.2 statsimi - Similarity Classification of Public Transit Stations

As part of our effort to find the best station similarity classifier, we developed *statsimi*[4], a Python tool that is able to extract station similarity ground truth data from OSM datasets and can be used to find station similarities either for individual stations via a built-in HTTP interface, or in bulk. It can be used to train and evaluate all the classifiers described in Section 2.8. For example, a classification model for Germany can be trained on an OSM file `germany.osm.bz2` as follows:

```
$ statsimi model --model_out model.mod --train germany.osm.bz2
```

A classification server using this model and listening on port 9999 can then for example be started like this:

```
$ statsimi http --model model.mod --http_port 9999
```

It serves a classification API at `/api` and a classification GUI at `/` (Figure 7.1).

An additional feature is to write *fix* files for OSM data. These are tab-separated files which contain for each station node or platform in a given input OSM file a list a confidence scores of how correct individual labels and the grouping of the nodes (via

[3] `https://travic.app/`
[4] `https://github.com/ad-freiburg/statsimi`

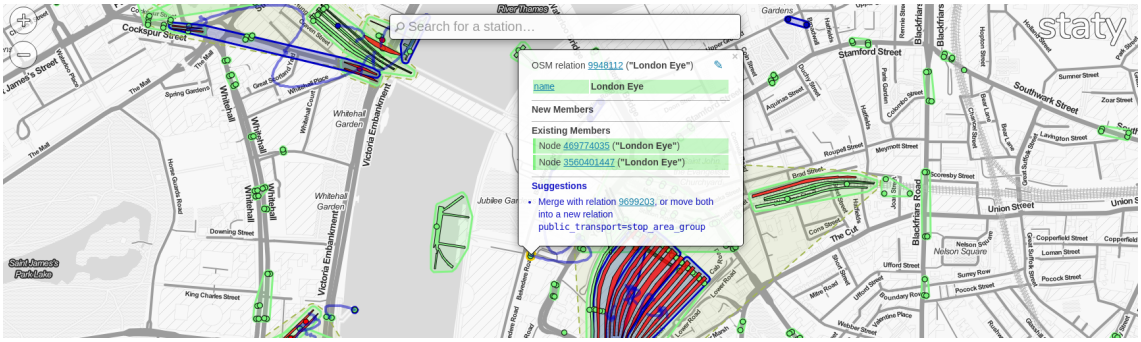


Figure 7.2: Our web tool *staty* for assessing the quality of public transit station tagging in OSM, based on fix files writting by *statsimi*.

OSM relations) are, based on similarity classification results. For confidence scores below a certain threshold, labeling and grouping suggestions are provided, which are obtained by re-clustering the station nodes based on the similarity scores, and translating the difference between the original grouping, and the re-clustered grouping into a set of suggestion rules. See [16] for details. A fix file for Germany using the model trained above can be generated as follows:

```
$ statsimi fix --model model.mod --fix_out de.fix --test germany.osm.bz2
```

7.3 staty - Quality Assessment of OSM Station Tagging

We have build a weekly updated web application called *staty*[5] which visualizes fix files generated by *statsimi* on a map. The app covers Western Europe, the United States, and Australia / New Zealand and features a heat-map overview to quickly see where errors in the OSM data have been found, and where suggestions are provided. Individual grouping suggestions are displayed on higher zoom levels, as shown in Figure 7.2.

7.4 topo - Map Construction for Line Graphs

For rendering transit maps, we have implemented a C++ software suite called *LOOM* (Line Ordering Optimized Maps). The line graph construction approach from Chapter 3 has been implemented in a tool called *topo*, which expects an arbitrary (possibly overlapping) line graph as a GeoJSON file and outputs a line graph which is ready for line ordering optimization, schematization, or direct rendering as a transit map. For converting GTFS schedule datasets into a preliminary line graph which can be fed to *topo*, a tool

[5] `https://staty.cs.uni-freiburg.de/`

gtfs2graph is additionally provided.

All tools in *LOOM* (except *gtfs2graph* and *transitmap*) expect a line graph on `stdin`, and write the result to `stdout`. For example, to extract a free line graph for the Freiburg tram network from the official GTFS data (placed in a folder `freiburg`), the following command suffices:

```
$ gtfs2graph -m tram freiburg | topo > freiburg.json
```

7.5 loom - Line-Ordering Optimization

All methods from Chapter 4 for finding the optimal line-ordering for a given input line graph have been implemented in the *loom* tool, which expects a line graph in `stdin` and writes the ordering-optimized line graph to `stdout`. A rich set of command line options may be used to specify for example the weights for line crossings and line separations, and the method used for optimization. By default, it internally uses the improved ILP approach with prior line graph simplification, solved with gurobi (if gurobi is not available, *loom* automatically falls back to first CBC, then GLPK). The following command finds the optimal line ordering for the Freiburg graph generated above:

```
$ loom < freiburg.json > freiburg.opt.json
```

For individual input datasets, one of the heuristic approaches described in Section 4.6 may be faster and sufficient. For example, the greedy lookahead heuristic can be used as follows:

```
$ loom -m greedy-lookahead < freiburg.json > freiburg.opt.json
```

7.6 octi - Fast Schematization of Line Graphs

The line graph schematization methods from Chapter 5 have been implemented in a tool called *octi*. It expects a line graph on `stdin` and outputs a schematic version of the line graph to `stdout`. For example, the line-ordering optimized line graph of the Freiburg tram network from above can be octilinearized as follows:

```
$ octi < freiburg.opt.json > freiburg.octi.json
```

The same tool can also create ortholinear, hexalinear, or orthoradial drawings:

```
$ octi -b porthoradial < freiburg.opt.json > freiburg.ortho.json
```

A GeoJSON file of *obstacles* (rivers, lakes, mountain ranges etc.) may be provided via the `--obstacles` parameter:

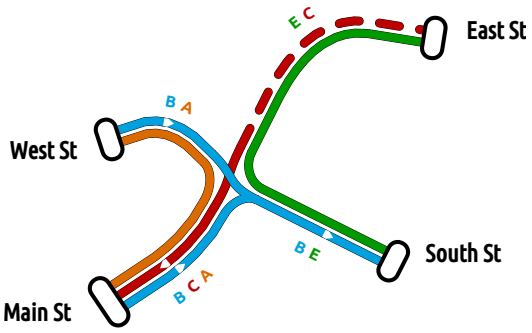


Figure 7.3: Rendering of a simple input line graph (with optimized line ordering) with labeling, directed line markers, and CSS styling (dashed line), generated by our tool *transitmap*.

```
$ octi --obstacles blackforest.json < freiburg.opt.json > freiburg.octi.json
```

To approximate the original input line courses to some degree p, it offers the parameter `--enf-geo-pen=<p>`.

7.7 transitmap - Transit Map Rendering Engine

Finally, the rendering and labeling approach described in Chapter 6 has been implemented in a tool called *transitmap*, which expects a line graph from `stdin`, and writes an SVG file to `stdout`. For example, the demo line graph `example2.json` provided with the LOOM software suite can be rendered as follows into an image `map.svg`:

```
$ transitmap < example2.json > map.svg
```

transitmap supports different line widths, different line spacing, and can render direction markers for directed lines (see Figure 7.3). As the output is an SVG graphic, it is easily customizable using CSS rules (see the dashed line in Figure 7.3). These rules can be embedded into the line graph file. Line graphs may also be provided in the DOT language, although this is still experimental.

transitmap is the last component in our toolchain used for generating (schematic) transit maps from raw schedule data. For example, an octilinear schematic transit map of the Freiburg tram network can be elegantly created from the raw GTFS schedule data in folder `freiburg` with the following command:

```
$ gtfs2graph -m tram freiburg | topo | octi | loom | transitmap > map.svg
```

As mentioned above, the strengths of this pipeline approach are that it can be easily extended, and individual components replaced. For example, a drop-in replacement for *octi* which schematizes the input line graph using the approach based on Bézier curves by Fink et al. [58] was implemented as a student project[6] (see Figure 7.4).

[6] `https://ad-blog.informatik.uni-freiburg.de/post/circular-transit-maps/`

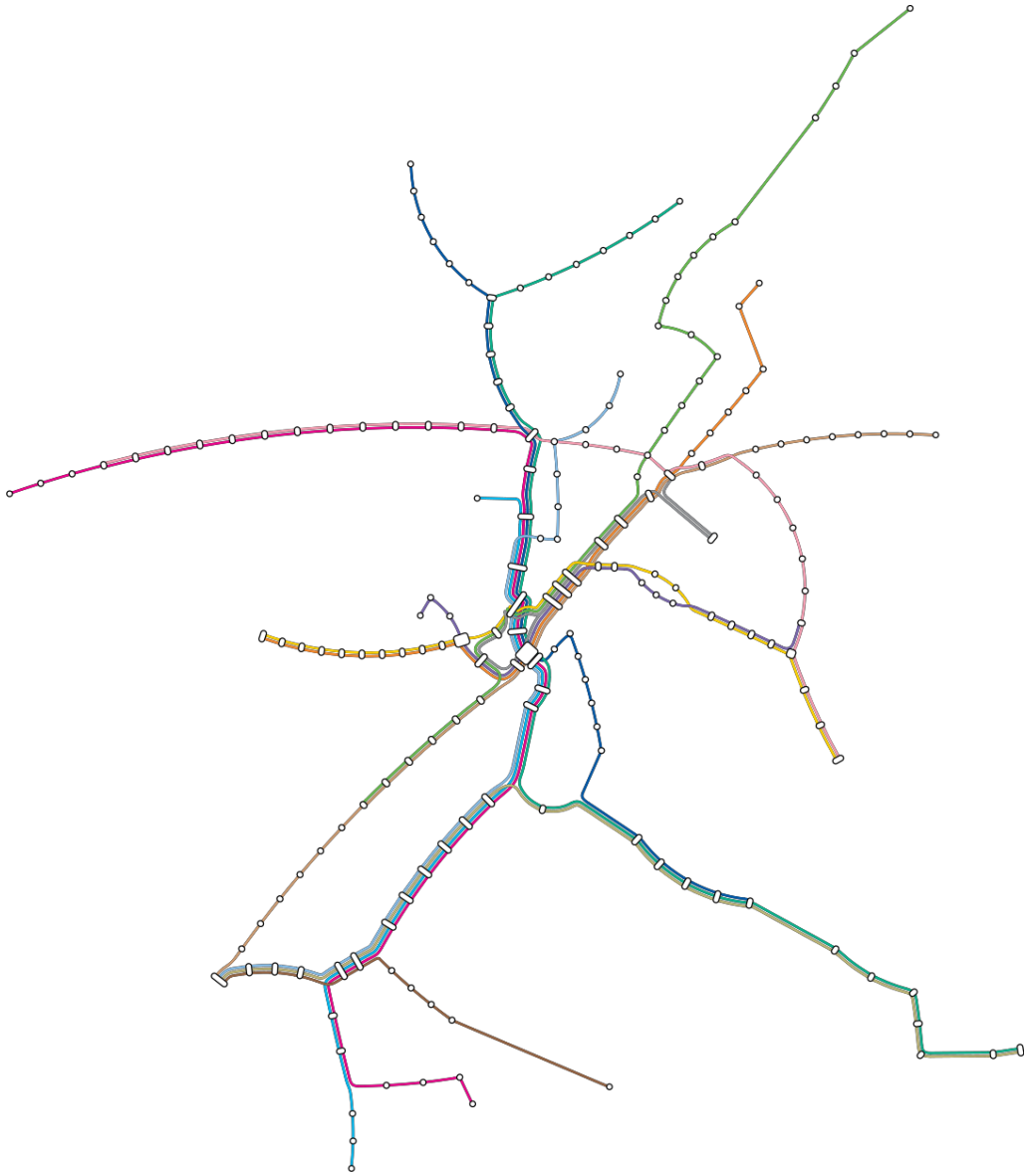


Figure 7.4: Curvilinear transit map of the 2016 Stuttgart light rail network generated using our pipeline and the schematization approach by Fink et al. [58]. The approach was implemented as a drop-in replacement for *octi* as part of a student project.

Chapter 8

Conclusions

This thesis described a full end-to-end approach to automatically render transit maps from schedule data. To achieve this, we identified six key problems from the fields of map-matching, similarity classification, map-construction, combinatorial optimization, graph drawing, and map rendering. We have developed algorithmic approaches to all of these problems and presented reference implementations in publicly available open-source tools.

Our map-matching method for finding missing shapes in schedule data adapted the state-of-the-art approach of using a hidden Markov model and also considered public-transit specific metadata from the transportation network. We demonstrated how both *intra-hop* and *inter-hop* turn restrictions can be naturally considered by routing on the edge-to-node dual of the transportation network graph. To better incorporate station metadata, we evaluated several methods for station similarity classification, among them a machine-learning based approach. A comprehensive evaluation showed the high quality of our approach: for example, for the Seattle and Vienna schedule data, we achieved accuracy scores of nearly 100%. Our method also seems to be more robust against imprecise input data than the standard method used in many routing and map-matching libraries. A major challenge was finding the missing shapes in reasonable time for very large input datasets. With a carefully designed set of speedup techniques, we achieved a running time of 8.5 minutes for the entire national schedule data of Germany. This did not include the parsing of the transportation network from OpenStreetMap (OSM) data, though, as our reference implementation *pfaedle* so far only supports the raw OSM XML format. As this is a major practical limitation, further work in *pfaedle* should extend the support to arbitrary OSM formats, and also offer an option to serialize the transportation network graph to disk, for fast later use. Our method so far also only utilizes the temporal information contained in the schedule data to a limited extent. While arrival and departure times in real-world schedule datasets are usually imprecise when compared to the actual travel time of the vehicle, they may nevertheless provide valuable information to prune unrealistic vehicle routes (for example, routes that are much too short). An ad-

ditional challenge for future research is thus to better consider this temporal data. One idea that seems promising is to generate more transition candidate paths (for example, by using methods for calculating alternative routes) and base the transition probabilities also on the scheduled travel time.

To prepare the shaped schedule data for later rendering, we developed a map construction method to transform an arbitrary line graph (a graph where edges are labeled with sets of transit lines, and nodes may be stations) into a *free* line graph, which is ready for rendering without line overlaps and with stations clustered using our station similarity classification approach. The input line graph may be directly created from schedule data, or even from a transportation network like contained in OSM. This method was an iteration on previous approaches. In this new approach, we did not track station nodes or line turn restrictions through the constructions process, but inferred them onto the final constructed graph. Although our method already produces satisfying results, we still consider it preliminary. During the evaluation, it emerged that clustering stations into single nodes was a bad idea - in some cases, there is no placement in the free line graph so that the clustered station node is served by all lines which originally served the clustered stops. The implementation of our approach so far uses preliminary line graphs obtained from (map-matched) schedule data. It would be highly interesting to extract such line graphs on a national, or even global level from OSM data, and test our construction approach on such datasets. This would enable us to create a global transit map from OSM data using the pipeline described in this work.

The combinatorial optimization problem of finding optimal permutations of the line labels for the edges in the free line graph was particularly challenging. Here, the goal was to produce as few line crossings in the final map as possible. We formulated the Metro Line Node Crossing Minimization problem (MLNCM), better suited for rendering than the classic Metro Line Crossing Minimization problem (MLCM), and developed variants in which the crossings are weighted (MLNCM-W), or which additionally consider the novel concept of line separations (MLNCM-S, MLNCM-WS), We proved the NP-hardness of MLNCM (and hence MLNCM-W and MLNCM-WS). This motivated the development of two integer linear programs (ILP) and several approximate approaches to tackle this problem. Using a sophisticated set of simplification rules which implicitly compute optimal partial line orderings and transform an input line graph into (typically many) components, we were able to optimize MLNCM-WS instances in under 0.6 seconds even for very large networks. We see potential in developing additional simplification rules. In particular, we outlined a whole class of new untangling rules, namely *induced* untangling rules.

To generate schematic transit maps, we developed a novel approach for a classic problem from graph drawing: metro map drawing. Our approach is based on finding metro map *images* on special octilinear grid graphs and can also consider obstacles (e.g. lakes or mountain ranges) or approximate the geographical course of the input lines. We

developed an ILP as well as a fast approximate approach to find the optimal images. For intermediate to large input networks (Chicago, Sydney, Berlin, Stuttgart), our approach was able to find high-quality octilinear maps in under 700 ms. Even for the highly complex London subway network, we arrived at a solution in 8.5 seconds, albeit with several constraint violations. We investigated the effect of *sparse* base grids on the quality and solution time and found that our idea of using an octilinear Hanan grid constructed from the input graph as a base grid greatly reduced the base grid size with only minimal impact on the solution quality. The impact on the solution time, however, was below our expectation. Regarding maps following other layouts, we showed that our approach can also create high-quality orthoradial or hexalinear maps. The main limitation of our approach so far is that station labels are not yet considered during the schematization. Although we achieved satisfying results by using a simple a posteriori labeling approach, we emphasize that further research is required here. It seems promising to reconsider the labeling approach from an early metro map drawing approach by Stott et al. [134] and apply it to our work. Another practically relevant problem that has received relatively little research interest so far is the automatic drawing of schematic transit maps with tariff zones. Here a promising approach seems to be to use the edges of the zone polygons as special edges in the input line graph.

Finally, we gave a practically oriented overview of how the free line graphs, either schematic or geographically accurate, and with optimal line ordering permutations, can be rendered into maps in an esthetically pleasing and informative way. This is in itself a fascinating topic of which we have merely scratched the surface, and there is recent and ongoing research to quantify the esthetic quality of metro maps. We consider it highly promising to use the approach described in this work to conduct a study on perceived map quality. For example, we may render transit maps using different layouts, different line orderings, and different styles for stations and segments, and do A/B testing on them.

We are also convinced that the software pipeline developed in the course of this work can serve as a solid framework for future research in the area of transit map drawing. In particular, all our tools are designed in such a way that they can be easily substituted by drop-in replacements.

List of Figures

List of Tables

Bibliography

[1] M. Ahmed, S. Karagiorgou, D. Pfoser, and C. Wenk, *Map Construction Algorithms*, Springer, 2015. https://doi.org/10.1007/978-3-319-25166-0.

[2] M. Ahmed and C. Wenk, Constructing street networks from GPS trajectories, in *Algorithms - ESA 2012 - 20th Annual European Symposium, Ljubljana, Slovenia, September 10-12, 2012. Proceedings*, *Lecture Notes in Computer Science* **7501**, Springer, 2012, pp. 60–71. https://doi.org/10.1007/978-3-642-33090-2_7.

[3] R. K. Ahuja, K. Mehlhorn, J. B. Orlin, and R. E. Tarjan, Faster algorithms for the shortest path problem, *J. ACM* **37** no. 2 (1990), 213–223. https://doi.org/10.1145/77600.77615.

[4] O. Aichholzer, F. Aurenhammer, D. Alberts, and B. Gärtner, A novel type of skeleton for polygons, *J. Univers. Comput. Sci.* **1** no. 12 (1995), 752–761. https://doi.org/10.3217/jucs-001-12-0752.

[5] H. Alt, A. Efrat, G. Rote, and C. Wenk, Matching planar maps, *J. Algorithms* **49** no. 2 (2003), 262–283. https://doi.org/10.1016/S0196-6774(03)00085-3.

[6] H. Alt and M. Godau, Computing the fréchet distance between two polygonal curves, *Int. J. Comput. Geom. Appl.* **5** (1995), 75–91. https://doi.org/10.1142/S0218195995000064.

[7] H. Alt and L. J. Guibas, Discrete geometric shapes: Matching, interpolation, and approximation, in *Handbook of Computational Geometry* (J. Sack and J. Urrutia, eds.), North Holland / Elsevier, 2000, pp. 121–153. https://doi.org/10.1016/b978-044482537-7/50004-8.

[8] H. Aly and M. Youssef, semMatch: Road semantics-based accurate map matching for challenging positioning data, in *Proceedings of the 23rd SIGSPATIAL International Conference on Advances in Geographic Information Systems, Bellevue, WA, USA, November 3-6, 2015*, ACM, 2015, pp. 5:1–5:10. https://doi.org/10.1145/2820783.2820824.

[9] S. Anand, S. Avelar, J. M. Ware, and M. Jackson, Automated schematic map production using simulated annealing and gradient descent approaches, in *GISRUK*, **7**, Citeseer, 2007, p. 2007.

[10] E. N. Argyriou, M. A. Bekos, M. Kaufmann, and A. Symvonis, On metro-line crossing minimization, *J. Graph Algorithms Appl.* **14** no. 1 (2010), 75–96. https://doi.org/10.7155/jgaa.00199.

[11] M. Asquith, J. Gudmundsson, and D. Merrick, An ILP for the metro-line crossing problem, in *Theory of Computing 2008. Proc. Fourteenth Computing: The Australasian Theory Symposium (CATS 2008), Wollongong, NSW, Australia, January 22-25, 2008. Proceedings*, *CRPIT* **77**, Australian Computer Society, 2008, pp. 49–56. Available at http://crpit.scem.westernsydney.edu.au/abstracts/CRPITV77Asquith.html.

[12] S. Avelar and M. Müller, *Generating topologically correct schematic maps*, Tech. report, ETH Zurich, 2000.

[13] L. Barth, B. Niedermann, I. Rutter, and M. Wolf, Towards a topology-shape-metrics framework for ortho-radial drawings, in *33rd International Symposium on Computational Geometry, SoCG 2017, July 4-7, 2017, Brisbane, Australia*, *LIPIcs* **77**, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017, pp. 14:1–14:16. https://doi.org/10.4230/LIPIcs.SoCG.2017.14.

[14] H. Bast and P. Brosi, Sparse map-matching in public transit networks with turn restrictions, in *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2018, Seattle, WA, USA, November 06-09, 2018*, ACM, 2018, pp. 480–483. https://doi.org/10.1145/3274895.3274957.

[15] H. Bast, P. Brosi, and M. Näther, Similarity classification of public transit stations, *CoRR* **abs/2012.15267** (2020). Available at https://arxiv.org/abs/2012.15267.

[16] H. Bast, P. Brosi, and M. Näther, staty: Quality assurance for public transit stations in OpenStreetMap, in *SIGSPATIAL '20: 28th International Conference on Advances in Geographic Information Systems, Seattle, WA, USA, November 3-6, 2020*, ACM, 2020, pp. 207–210. https://doi.org/10.1145/3397536.3422342.

[17] H. Bast, P. Brosi, and S. Storandt, Real-time movement visualization of public transit data, in *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, Dallas/Fort Worth, TX, USA, November 4-7, 2014*, ACM, 2014, pp. 331–340. https://doi.org/10.1145/2666310.2666404.

[18] H. Bast, P. Brosi, and S. Storandt, TRAVIC: a visualization client for public transit data, in *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, Dallas/Fort Worth, TX, USA, November 4-7, 2014*, ACM, 2014, pp. 561–564. https://doi.org/10.1145/2666310.2666369.

[19] H. Bast, P. Brosi, and S. Storandt, Efficient generation of geographically accurate transit maps, in *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2018, Seattle, WA, USA, November 06-09, 2018*, ACM, 2018, pp. 13–22. https://doi.org/10.1145/3274895.3274955.

[20] H. Bast, P. Brosi, and S. Storandt, Efficient generation of geographically accurate transit maps, *ACM Trans. Spatial Algorithms Syst.* **5** no. 4 (2019), 25:1–25:36. https://doi.org/10.1145/3337790.

[21] H. Bast, P. Brosi, and S. Storandt, Metro maps on octilinear grid graphs, *Comput. Graph. Forum* **39** no. 3 (2020), 357–367. https://doi.org/10.1111/cgf.13986.

[22] H. Bast, P. Brosi, and S. Storandt, Metro maps on flexible base grids, in *Proceedings of the 17th International Symposium on Spatial and Temporal Databases, SSTD 2021, Virtual Event, USA, August 23-25, 2021*, ACM, 2021, pp. 12–22. https://doi.org/10.1145/3469830.3470899.

[23] H. Bast, D. Delling, A. V. Goldberg, M. Müller-Hannemann, T. Pajor, P. Sanders, D. Wagner, and R. F. Werneck, Route planning in transportation networks, in *Algorithm Engineering - Selected Results and Surveys* (L. Kliemann and P. Sanders, eds.), *Lecture Notes in Computer Science* **9220**, 2016, pp. 19–80. https://doi.org/10.1007/978-3-319-49487-6_2.

[24] T. Behr, T. C. van Dijk, A. Forsch, J.-H. Haunert, and S. Storandt, Map matching for semi-restricted trajectories, in *11th International Conference on Geographic Information Science (GIScience 2021)-Part II*, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021. https://doi.org/10.4230/LIPIcs.GIScience.2021.II.12.

[25] M. A. Bekos, M. Kaufmann, K. Potika, and A. Symvonis, Line crossing minimization on metro maps, in *Graph Drawing, 15th International Symposium, GD 2007, Sydney, Australia, September 24-26, 2007. Revised Papers*, *Lecture Notes in Computer Science* **4875**, Springer, 2007, pp. 231–242. https://doi.org/10.1007/978-3-540-77537-9_24.

[26] M. Benkert, M. Nöllenburg, T. Uno, and A. Wolff, Minimizing intra-edge crossings in wiring diagrams and public transportation maps, in *Graph Drawing, 14th International Symposium, GD 2006, Karlsruhe, Germany, September 18-20, 2006. Revised Papers*, *Lecture Notes in Computer Science* **4372**, Springer, 2006, pp. 270–281. https://doi.org/10.1007/978-3-540-70904-6_27.

[27] J. Biagioni and J. Eriksson, Inferring road maps from global positioning system traces: Survey and comparative evaluation, *Transportation research record* **2291** no. 1 (2012), 61–71. https://doi.org/10.3141/2291-08.

[28] J. Biagioni and J. Eriksson, Map inference in the face of noise and disparity, in *SIGSPATIAL 2012 International Conference on Advances in Geographic Information Systems, SIGSPATIAL'12, Redondo Beach, CA, USA, November 7-9, 2012*, ACM, 2012, pp. 79–88. https://doi.org/10.1145/2424321.2424333.

[29] M. Bilenko, R. J. Mooney, W. W. Cohen, P. Ravikumar, and S. E. Fienberg, Adaptive name matching in information integration, *IEEE Intell. Syst.* **18** no. 5 (2003), 16–23. https://doi.org/10.1109/MIS.2003.1234765.

[30] S. Brakatsoulas, D. Pfoser, R. Salas, and C. Wenk, On map-matching vehicle tracking data, in *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, ACM, 2005, pp. 853–864. Available at http://www.vldb.org/archives/website/2005/program/paper/fri/p853-brakatsoulas.pdf.

[31] P. Brosi and U. Müller, Automatisierte Erstellung von fahrplanreferenzierten ÖV-Netzen, *AGIT Journal Angew. Geoinformatik* **1** (2015), 122–129. https://doi.org/10.14627/537557017.

[32] M. E. Buchin, *On the computability of the Fréchet distance between triangulated surfaces*, Ph.D. thesis, FU Berlin, 2007. https://doi.org/10.17169/refubium-6111.

[33] S. Cabello, M. de Berg, S. van Dijk, M. J. van Kreveld, and T. Strijk, Schematization of road networks, in *Proceedings of the Seventeenth Annual Symposium on Computational Geometry, Medford, MA, USA, June 3-5, 2001*, ACM, 2001, pp. 33–39. https://doi.org/10.1145/378583.378609.

[34] F. Cacciola, A survey of polygon offseting strategies, http://fcacciola.50webs.com/Offseting%20Methods.htm, Accessed: 2022-03-07.

[35] L. Cao and J. Krumm, From GPS traces to a routable road map, in *17th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems, ACM-GIS 2009, November 4-6, 2009, Seattle, Washington, USA, Proceedings*, ACM, 2009, pp. 3–12. https://doi.org/10.1145/1653771.1653776.

[36] G. M. Chaikin, An algorithm for high-speed curve generation, *Comput. Graph. Image Process.* **3** no. 4 (1974), 346–349. https://doi.org/10.1016/0146-664X(74)90028-8.

[37] P. Chao, Y. Xu, W. Hua, and X. Zhou, A survey on map-matching algorithms, in *Databases Theory and Applications - 31st Australasian Database Conference, ADC 2020, Melbourne, VIC, Australia, February 3-7, 2020, Proceedings*, *Lecture Notes in Computer Science* **12008**, Springer, 2020, pp. 121–133. https://doi.org/10.1007/978-3-030-39469-1_10.

[38] D. Chivers and P. Rodgers, Octilinear force-directed layout with mental map preservation for schematic diagrams, in *Diagrammatic Representation and Inference - 8th International Conference, Diagrams 2014, Melbourne, VIC, Australia, July 28 - August 1, 2014. Proceedings*, *Lecture Notes in Computer Science* **8578**, Springer, 2014, pp. 1–8. https://doi.org/10.1007/978-3-662-44043-8_1.

[39] J. Christensen, J. Marks, and S. M. Shieber, An empirical study of algorithms for point-feature label placement, *ACM Trans. Graph.* **14** no. 3 (1995), 203–232. https://doi.org/10.1145/212332.212334.

[40] J. Chuzhoy and D. H. K. Kim, On approximating node-disjoint paths in grids, in *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, APPROX/RANDOM 2015, August 24-26, 2015, Princeton, NJ, USA*, *LIPIcs* **40**, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015, pp. 187–211. https://doi.org/10.4230/LIPIcs.APPROX-RANDOM.2015.187.

[41] W. W. Cohen, P. Ravikumar, and S. E. Fienberg, A comparison of string distance metrics for name-matching tasks, in *IJCAI 2003, Acapulco, Mexico. Proceedings.*, 2003, pp. 73–78.

[42] D. Conte, P. Foggia, C. Sansone, and M. Vento, Thirty years of graph matching in pattern recognition, *Int. J. Pattern Recognit. Artif. Intell.* **18** no. 3 (2004), 265–298. https://doi.org/10.1142/S0218001404003228.

[43] G. B. Dantzig, Maximization of a linear function of variables subject to linear inequalities, *Activity analysis of production and allocation* **13** (1951), 339–347.

[44] J. J. Davies, A. R. Beresford, and A. Hopper, Scalable, distributed, real-time map generation, *IEEE Pervasive Comput.* **5** no. 4 (2006), 47–54. https://doi.org/10.1109/MPRV.2006.83.

[45] A. Degani, A tale of two maps: Analysis of the London underground "diagram", *Ergonomics in Design* **21** no. 3 (2013), 7–16.

[46] Y. Deng, A. Luo, J. Liu, and Y. Wang, Point of interest matching between different geospatial datasets, *Int. J. Geogr. Inf. Sci.* **8** no. 10 (2019), 435. https://doi.org/10.3390/ijgi8100435.

[47] J. Dibbelt, B. Strasser, and D. Wagner, Customizable contraction hierarchies, in *International Symposium on Experimental Algorithms*, Springer, 2014, pp. 271–282. https://doi.org/10.1007/978-3-319-07959-2_23.

[48] T. C. van Dijk, A. van Goethem, J. Haunert, W. Meulemans, and B. Speckmann, Map schematization with circular arcs, in *Geographic Information Science - 8th International Conference, GIScience 2014, Vienna, Austria, September 24-26, 2014. Proceedings*, *Lecture Notes in Computer Science* **8728**, Springer, 2014, pp. 1–17. https://doi.org/10.1007/978-3-319-11593-1_1.

[49] T. C. van Dijk and D. Lutz, Realtime linear cartograms and metro maps, in *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, SIGSPATIAL 2018, Seattle, WA, USA, November 06-09, 2018*, ACM, 2018, pp. 488–491. https://doi.org/10.1145/3274895.3274959.

[50] E. W. Dijkstra and others, A note on two problems in connexion with graphs, *Numerische mathematik* **1** no. 1 (1959), 269–271. https://doi.org/10.1007/BF01386390.

[51] S. Edelkamp and S. Schrödl, Route planning and map inference with global positioning traces, in *Computer Science in Perspective, Essays Dedicated to Thomas Ottmann*, *Lecture Notes in Computer Science* **2598**, Springer, 2003, pp. 128–151. https://doi.org/10.1007/3-540-36477-3_10.

[52] J. Eisner, S. Funke, A. Herbst, A. Spillner, and S. Storandt, Algorithms for matching and predicting trajectories, in *Proceedings of the Thirteenth Workshop on Algorithm Engineering and Experiments, ALENEX 2011, Holiday Inn San Francisco Golden Gateway, San Francisco, California, USA, January 22, 2011*, SIAM, 2011, pp. 84–95. https://doi.org/10.1137/1.9781611972917.9.

[53] T. Eiter and H. Mannila, *Computing discrete Fréchet distance*, Tech. report, Christian Doppler Laboratory for Expert Systems, TU Vienna, Austria, 1994.

[54] D. Elroi, Designing a network line-map schematization software enhancement package, in *Proc. 8th Ann. ESRI User Conference*, 1988.

[55] D. Elroi, Gis and schematic maps: A new symbiotic relationship, in *Proc. GIS/LIS*, **88**, 1988.

[56] A. Fathi and J. Krumm, Detecting road intersections from GPS traces, in *Geographic Information Science, 6th International Conference, GIScience 2010, Zurich, Switzerland, September 14-17, 2010. Proceedings*, *Lecture Notes in Computer Science* **6292**, Springer, 2010, pp. 56–69. https://doi.org/10.1007/978-3-642-15300-6_5.

[57] A. A. Ferreira, M. A. Gonçalves, and A. H. F. Laender, A brief survey of automatic methods for author name disambiguation, *SIGMOD Record* **41** no. 2 (2012), 15–26. https://doi.org/10.1145/2350036.2350040.

[58] M. Fink, H. J. Haverkort, M. Nöllenburg, M. J. Roberts, J. Schuhmann, and A. Wolff, Drawing metro maps using bézier curves, in *Graph Drawing - 20th International Symposium, GD 2012, Redmond, WA, USA, September 19-21, 2012, Revised Selected Papers*, *Lecture Notes in Computer Science* **7704**, Springer, 2012, pp. 463–474. https://doi.org/10.1007/978-3-642-36763-2_41.

[59] M. Fink and S. Pupyrev, Metro-line crossing minimization: Hardness, approximations, and tractable cases, in *Graph Drawing - 21st International Symposium, GD 2013, Bordeaux, France, September 23-25, 2013, Revised Selected Papers*, *Lecture Notes in Computer Science* **8242**, Springer, 2013, pp. 328–339. https://doi.org/10.1007/978-3-319-03841-4_29.

[60] M. Fink and S. Pupyrev, Ordering metro lines by block crossings, in *Mathematical Foundations of Computer Science 2013 - 38th International Symposium, MFCS 2013, Klosterneuburg, Austria, August 26-30, 2013. Proceedings*, *Lecture Notes in Computer Science* **8087**, Springer, 2013, pp. 397–408. https://doi.org/10.1007/978-3-642-40313-2_36.

[61] M. Formann and F. Wagner, A packing problem with applications to lettering of maps, in *Proceedings of the Seventh Annual Symposium on Computational Geometry, North Conway, NH, USA, , June 10-12, 1991*, ACM, 1991, pp. 281–288. https://doi.org/10.1145/109648.109680.

[62] F. Frank, M. Kaufmann, S. G. Kobourov, T. Mchedlidze, S. Pupyrev, T. Ueckerdt, and A. Wolff, Using the metro-map metaphor for drawing hypergraphs, in *SOFSEM 2021: Theory and Practice of Computer Science - 47th International Conference on Current Trends in Theory and Practice of Computer Science, SOFSEM 2021, Bolzano-Bozen, Italy, January 25-29, 2021, Proceedings*, *Lecture Notes in Computer Science* **12607**, Springer, 2021, pp. 361–372. https://doi.org/10.1007/978-3-030-67731-2_26.

[63] M. Fréchet, Sur quelques points du calcul fonctionnel, *Rendiconti del Circolo Matematico di Palermo (1884-1940)* **22** no. 1 (1906), 1–72. https://doi.org/10.1007/BF03018603.

[64] M. L. Fredman and R. E. Tarjan, Fibonacci heaps and their uses in improved network optimization algorithms, *J. ACM* **34** no. 3 (1987), 596–615. https://doi.org/10.1145/28869.28874.

[65] X. Gao, B. Xiao, D. Tao, and X. Li, A survey of graph edit distance, *Pattern Anal. Appl.* **13** no. 1 (2010), 113–129. https://doi.org/10.1007/s10044-008-0141-y.

[66] M. R. Garey and D. S. Johnson, The rectilinear steiner tree problem is np-complete, *SIAM Journal on Applied Mathematics* **32** no. 4 (1977), 826–834. https://doi.org/10.1137/0132071.

[67] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*, W. H. Freeman & Co., USA, 1990.

[68] K. Garland, *Mr. Beck's Underground Map*, Capital Transport Pub, 1994.

[69] G. Gati, Further annotated bibliography on the isomorphism disease, *J. Graph Theory* **3** no. 2 (1979), 95–109. https://doi.org/10.1002/jgt.3190030202.

[70] R. Geisberger, P. Sanders, D. Schultes, and D. Delling, Contraction hierarchies: Faster and simpler hierarchical routing in road networks, in *Experimental Algorithms, 7th International Workshop, WEA 2008, Provincetown, MA, USA, May 30-June 1, 2008, Proceedings* (C. C. McGeoch, ed.), *Lecture Notes in Computer Science* **5038**, Springer, 2008, pp. 319–333. https://doi.org/10.1007/978-3-540-68552-4_24.

[71] R. Geisberger and C. Vetter, Efficient routing in road networks with turn costs, in *International Symposium on Experimental Algorithms*, Springer, 2011, pp. 100–111. https://doi.org/10.1007/978-3-642-20662-7_9.

[72] A. Gemsa, B. Niedermann, and M. Nöllenburg, A unified model and algorithms for temporal map labeling, *Algorithmica* **82** no. 10 (2020), 2709–2736. https://doi.org/10.1007/s00453-020-00694-7.

[73] M. Goldapp, Approximation of circular arcs by cubic polynomials, *Computer Aided Geometric Design* **8** no. 3 (1991), 227–238. https://doi.org/10.1016/0167-8396(91)90007-X.

[74] A. V. Goldberg and C. Harrelson, Computing the shortest path: A^* search meets graph theory, in *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2005, Vancouver, British Columbia, Canada, January 23-25, 2005*, SIAM, 2005, pp. 156–165. Available at http://dl.acm.org/citation.cfm?id=1070432.1070455.

[75] P. Groeneveld, Wire ordering for detailed routing, *IEEE Des. Test* **6** no. 6 (1989), 6–17. https://doi.org/10.1109/54.41670.

[76] GTFS Reference, https://developers.google.com/transit/gtfs/reference, Accessed: 2022-02-25.

[77] Hafas Rohdaten Format (HRDF), https://opentransportdata.swiss/de/cookbook/hafas-rohdaten-format-hrdf/, Accessed: 2022-02-22.

[78] H. Han, C. L. Giles, H. Zha, C. Li, and K. Tsioutsiouliklis, Two supervised learning approaches for name disambiguation in author citations, in *ACM/IEEE JCDL 2004, Tucson, AZ, USA. Proceedings.*, 2004, pp. 296–305. https://doi.org/10.1145/996350.996419.

[79] M. Hanan, On Steiner's problem with rectilinear distance, *SIAM Journal on Applied Mathematics* **14** no. 2 (1966), 255–265.

[80] F. Harary and R. Z. Norman, Some properties of line digraphs, *Rendiconti del Circolo Matematico di Palermo* **9** no. 2 (1960), 161–168. https://doi.org/10.1007/BF02854581.

[81] P. E. Hart, N. J. Nilsson, and B. Raphael, A formal basis for the heuristic determination of minimum cost paths, *IEEE transactions on Systems Science and Cybernetics* **4** no. 2 (1968), 100–107. https://doi.org/10.1109/TSSC.1968.300136.

[82] O. Hoell, Netzpläne der Berliner S-Bahn, https://www.s-bahn-galerie.de/S_Bahn_Berlin/xPlan_Bln/Plan_S_B.htm, Accessed: 2022-02-25.

[83] D. Holten and J. J. van Wijk, Force-directed edge bundling for graph visualization, in *Proceedings of the 11th Eurographics / IEEE - VGTC Conference on Visualization*, *EuroVis '09*, The Eurographs Association, 2009, pp. 983–998. https://doi.org/10.1111/j.1467-8659.2009.01450.x.

[84] S. Hong, D. Merrick, and H. A. D. do Nascimento, The metro map layout problem, in *Graph Drawing, 12th International Symposium, GD 2004, New York, NY, USA, September 29 - October 2, 2004, Revised Selected Papers*, *Lecture Notes in Computer Science* **3383**, Springer, 2004, pp. 482–491. https://doi.org/10.1007/978-3-540-31843-9_50.

[85] S.-H. Hong, D. Merrick, and H. A. do Nascimento, Automatic visualisation of metro maps, *Journal of Visual Languages & Computing* **17** no. 3 (2006), 203–224. https://doi.org/10.1016/j.jvlc.2005.09.001.

[86] B. Hummel, *Map Matching for Vehicle Guidance*, CRC Press, 2006.

[87] I. Hussain and S. Asghar, A survey of author name disambiguation techniques: 2010-2016, *Knowledge Eng. Review* **32** (2017), e22. https://doi.org/10.1017/S0269888917000182.

[88] A. Itai, C. H. Papadimitriou, and J. L. Szwarcfiter, Hamilton paths in grid graphs, *SIAM Journal on Computing* **11** no. 4 (1982), 676–686. https://doi.org/10.1137/0211056.

[89] M. A. Jaro, Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida, *JASA* **84** no. 406 (1989), 414–420. https://doi.org/10.1080/01621459.1989.10478785.

[90] S. Karagiorgou and D. Pfoser, On vehicle tracking data-based road network generation, in *SIGSPATIAL 2012 International Conference on Advances in Geographic Information Systems, SIGSPATIAL'12, Redondo Beach, CA, USA, November 7-9, 2012*, ACM, 2012, pp. 89–98. https://doi.org/10.1145/2424321.2424334.

[91] R. M. Karp, On the computational complexity of combinatorial problems, *Networks* **5** no. 1 (1975), 45–68. https://doi.org/10.1002/net.1975.5.1.45.

[92] A. J. Kent, When topology trumped topography: Celebrating 90 years of Beck's underground map, 2021, pp. 1–12.

[93] L. G. Khachiyan, A polynomial algorithm in linear programming, in *Doklady Akademii Nauk*, **244**, Russian Academy of Sciences, 1979, pp. 1093–1096.

[94] S. Kirkpatrick, C. Gelatt, and M. Vecchi, Optimization by simulated annealing, *Science (New York, N.Y.)* **220** (1983), 671–80. https://doi.org/10.1126/science.220.4598.671.

[95] V. Klee and G. J. Minty, How good is the simplex algorithm, *Inequalities* **3** no. 3 (1972), 159–175.

[96] W. R. Knight, A computer method for calculating Kendall's tau with ungrouped data, *Journal of the American Statistical Association* **61** no. 314 (1966), 436–439. https://doi.org/10.1080/01621459.1966.10480879.

[97] M. R. Kramer, The complexity of wirerouting and finding minimum area layouts for arbitrary VLSI circuits, *Advances in computing research* **2** (1984), 129–146.

[98] P. Kramer and J. Van Leeuwen, *Wire routing in NP-complete*, Department of Computer Science, University of Utrecht Utrecht, The Netherlands, 1982.

[99] A. S. LaPaugh and R. L. Rivest, The subgraph homeomorphism problem, *Journal of Computer and System Sciences* **20** no. 2 (1980), 133–149. https://doi.org/10.1016/0022-0000(80)90057-4.

[100] J. Leskovec, A. Rajaraman, and J. D. Ullman, *Mining of Massive Datasets, 2nd Ed*, Cambridge University Press, 2014. https://doi.org/10.1017/CBO9781139924801.

[101] Z. Li and W. Dong, A stroke-based method for automated generation of schematic network maps, *Int. J. Geogr. Inf. Sci.* **24** no. 11 (2010), 1631–1647. https://doi.org/10.1080/13658811003766936.

[102] S. Liu, Y. Chu, H. Hu, J. Feng, and X. Zhu, Top-k spatio-textual similarity search, in *WAIM 2014, Macau, China, June 16-18, 2014. Proceedings*, **8485**, Springer, 2014, pp. 602–614. https://doi.org/10.1007/978-3-319-08010-9_65.

[103] X. Liu, J. Biagioni, J. Eriksson, Y. Wang, G. Forman, and Y. Zhu, Mining large-scale, sparse GPS traces for map inference: comparison of approaches, in *The 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '12, Beijing, China, August 12-16, 2012* (Q. Yang, D. Agarwal, and J. Pei, eds.), ACM, 2012, pp. 669–677. https://doi.org/10.1145/2339530.2339637.

[104] Y. Lou, C. Zhang, Y. Zheng, X. Xie, W. Wang, and Y. Huang, Map-matching for low-sampling-rate GPS trajectories, in *17th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems, ACM-GIS 2009, November 4-6, 2009, Seattle, Washington, USA, Proceedings*, ACM, 2009, pp. 352–361. https://doi.org/10.1145/1653771.1653820.

[105] J. F. Lynch, The equivalence of theorem proving and the interconnection problem, *ACM SIGDA Newsletter* **5** no. 3 (1975), 31–36. https://doi.org/10.1145/1061425.1061430.

[106] D. Merrick and J. Gudmundsson, Path simplification for metro map layout, in *Graph Drawing, 14th International Symposium, GD 2006, Karlsruhe, Germany, September 18-20, 2006. Revised Papers*, *Lecture Notes in Computer Science* **4372**, Springer, 2006, pp. 258–269. https://doi.org/10.1007/978-3-540-70904-6_26.

[107] T. Milea, O. Schrijvers, K. Buchin, and H. J. Haverkort, Shortest-paths preserving metro maps, in *Graph Drawing - 19th International Symposium, GD 2011, Eindhoven, The Netherlands, September 21-23, 2011, Revised Selected Papers*, *Lecture Notes in Computer Science* **7034**, Springer, 2011, pp. 445–446. https://doi.org/10.1007/978-3-642-25878-7_45.

[108] J. Mondzech and M. Sester, Quality analysis of OpenStreetMap data based on application needs, *Cartogr. Int. J. Geogr. Inf. Geovisualization* **46** no. 2 (2011), 115–125. https://doi.org/10.3138/carto.46.2.115.

[109] P. Newson and J. Krumm, Hidden Markov map matching through noise and sparseness, in *17th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems, ACM-GIS 2009, November 4-6, 2009, Seattle, Washington, USA, Proceedings*, ACM, 2009, pp. 336–343. https://doi.org/10.1145/1653771.1653818.

[110] E. J. Newton and M. J. Roberts, Concentric circles maps: Data and implications, *Proceedings of the Theory and Application of Diagrams (Edinburgh, UK, 2018)* **17** (2018), 24.

[111] G. Neyer, Line simplification with restricted orientations, in *Algorithms and Data Structures, 6th International Workshop, WADS '99, Vancouver, British Columbia, Canada, August 11-14, 1999, Proceedings*, *Lecture Notes in Computer Science* **1663**, Springer, 1999, pp. 13–24. https://doi.org/10.1007/3-540-48447-7_2.

[112] S. Nickel and M. Nöllenburg, Drawing k-linear metro maps, *CoRR* **abs/1904.03039** (2019). Available at http://arxiv.org/abs/1904.03039.

[113] B. Niedermann and I. Rutter, An integer-linear program for bend-minimization in ortho-radial drawings, in *Graph Drawing and Network Visualization - 28th International Symposium, GD 2020, Vancouver, BC, Canada, September 16-18, 2020, Revised Selected Papers* (D. Auber and P. Valtr, eds.), *Lecture Notes in Computer Science* **12590**, Springer, 2020, pp. 235–249. https://doi.org/10.1007/978-3-030-68766-3_19.

[114] B. Niedermann, I. Rutter, and M. Wolf, Efficient algorithms for ortho-radial graph drawing, in *35th International Symposium on Computational Geometry, SoCG 2019, June 18-21, 2019, Portland, Oregon, USA*, *LIPIcs* **129**, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019, pp. 53:1–53:14. https://doi.org/10.4230/LIPIcs.SoCG.2019.53.

[115] M. Nöllenburg, *Automated drawing of metro maps*, Universität Karlsruhe, Fakultät für Informatik, 2005.

[116] M. Nöllenburg, An improved algorithm for the metro-line crossing minimization problem, in *Graph Drawing, 17th International Symposium, GD 2009, Chicago, IL, USA, September 22-25, 2009. Revised Papers*, *Lecture Notes in Computer Science* **5849**, Springer, 2009, pp. 381–392. https://doi.org/10.1007/978-3-642-11805-0_36.

[117] M. Nöllenburg and A. Wolff, A mixed-integer program for drawing high-quality metro maps, in *Graph Drawing, 13th International Symposium, GD 2005, Limerick, Ireland, September 12-14, 2005, Revised Papers*, *Lecture Notes in Computer Science* **3843**, Springer, 2005, pp. 321–333. https://doi.org/10.1007/11618058_29.

[118] M. Nöllenburg and A. Wolff, Drawing and labeling high-quality metro maps by mixed-integer programming, *IEEE Trans. Vis. Comput. Graph.* **17** no. 5 (2011), 626–641. https://doi.org/10.1109/TVCG.2010.81.

[119] T. Osogami and R. Raymond, Map matching with inverse reinforcement learning, in *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, IJCAI/AAAI, 2013, pp. 2547–2553. Available at http://www.aaai.org/ocs/index.php/IJCAI/IJCAI13/paper/view/6184.

[120] M. Pincus, A monte carlo method for the approximate solution of certain types of constrained optimization problems, *Operations research* **18** no. 6 (1970), 1225–1228. https://doi.org/10.1287/opre.18.6.1225.

[121] O. Pink and B. Hummel, A statistical approach to map matching using road network geometry, topology and vehicular motion constraints, in *11th International IEEE Conference on Intelligent Transportation Systems, ITSC 2008, Beijing, China, 12-15 October 2008*, IEEE, 2008, pp. 862–867. https://doi.org/10.1109/ITSC.2008.4732697.

[122] F. Poletti, P. M. Bösch, F. Ciari, and K. W. Axhausen, Public transit route mapping for large-scale multimodal networks, *ISPRS International Journal of Geo-Information* **6** no. 9 (2017), 268. https://doi.org/10.3390/ijgi6090268.

[123] S. Pupyrev, L. Nachmanson, S. Bereg, and A. E. Holroyd, Edge routing with ordered bundles, *Comput. Geom.* **52** (2016), 18–33. https://doi.org/10.1016/j.comgeo.2015.10.005.

[124] L. Rabiner and B. Juang, An introduction to hidden Markov models, *ieee assp magazine* **3** no. 1 (1986), 4–16.

[125] A. Riškus, Approximation of a cubic bézier curve by circular arcs and vice versa, *Information Technology and control* **35** no. 4 (2006).

[126] N. Robertson and P. D. Seymour, Graph minors. xiii. the disjoint paths problem, *Journal of combinatorial theory, Series B* **63** no. 1 (1995), 65–110. https://doi.org/10.1006/jctb.1995.1006.

[127] S. Rogers, P. Langley, and C. Wilson, Mining GPS data to augment road models, in *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 15-18, 1999*, ACM, 1999, pp. 104–113. https://doi.org/10.1145/312129.312208.

[128] E. Safra, Y. Kanza, Y. Sagiv, C. Beeri, and Y. Doytsher, Location-based algorithms for finding sets of corresponding objects over several geo-spatial data sets, *Int. J. Geogr. Inf. Sci.* **24** no. 1 (2010), 69–106. https://doi.org/10.1080/13658810802275560.

[129] T. Scheffler, R. Schirru, and P. Lehmann, Matching points of interest from different social networking sites, in *KI 2012, Saarbrücken, Germany, September 24-27, 2012. Proceedings*, **7526**, Springer, 2012, pp. 245–248. https://doi.org/10.1007/978-3-642-33347-7_24.

[130] S. Schrödl, K. Wagstaff, S. Rogers, P. Langley, and C. Wilson, Mining GPS traces for map refinement, *Data Min. Knowl. Discov.* **9** no. 1 (2004), 59–87. https://doi.org/10.1023/B:DAMI.0000026904.74892.89.

[131] M. Sniedovich, Dijkstra's algorithm revisited: the dynamic programming connexion, *Control and cybernetics* **35** (2006), 599–620.

[132] M. Srivatsa, R. Ganti, J. Wang, and V. Kolar, Map matching: Facts and myths, in *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, ACM, 2013, pp. 484–487. https://doi.org/10.1145/2525314.2525466.

[133] J. M. Stott and Rodgers, Metro map layout using multicriteria optimization, in *Proceedings. Eighth International Conference on Information Visualisation, 2004. IV 2004.*, 2004, pp. 355–362. https://doi.org/10.1109/IV.2004.1320168.

[134] J. M. Stott, P. Rodgers, J. C. Martinez-Ovando, and S. G. Walker, Automatic metro map layout using multicriteria optimization, *IEEE Trans. Vis. Comput. Graph.* **17** no. 1 (2011), 101–114. https://doi.org/10.1109/TVCG.2010.24.

[135] Timetable information data feed interface specification, https://www.raildeliverygroup.com/files/Publications/services/rsp/RSPS5046_timetable_information_data_feed_interface_specification.pdf, Accessed: 2022-02-22.

[136] R. Tamassia, *Handbook of graph drawing and visualization*, CRC press, 2013. https://doi.org/10.1201/b15385.

[137] P. Ti and Z. Li, Generation of schematic network maps with automated detection and enlargement of congested areas, *Int. J. Geogr. Inf. Sci.* **28** no. 3 (2014), 521–540. https://doi.org/10.1080/13658816.2013.855313.

[138] A. Tibaut, Representing timetables in NeTEx, https://www.netex-cen.eu/wp-content/uploads/2015/12/09.NeTEx-Timetable-WhitePaper_1.05.pdf, Accessed: 2022-03-07.

[139] VDV-Standardschnittstelle Liniennetz/Fahrplan, https://www.vdv.de/452-sdsv15.pdfx, Accessed: 2022-02-24.

[140] A. Viterbi, Error bounds for convolutional codes and an asymptotically optimum decoding algorithm, *IEEE Transactions on Information Theory* **13** no. 2 (1967), 260–269. https://doi.org/10.1109/TIT.1967.1054010.

[141] D. Wagner and T. Willhalm, Speed-up techniques for shortest-path computations, in *STACS 2007, 24th Annual Symposium on Theoretical Aspects of Computer Science, Aachen, Germany, February 22-24, 2007, Proceedings* (W. Thomas and P. Weil, eds.), *Lecture Notes in Computer Science* **4393**, Springer, 2007, pp. 23–36. https://doi.org/10.1007/978-3-540-70918-3_3.

[142] Y. Wang and M. Chi, Focus+context metro maps, *IEEE Trans. Vis. Comput. Graph.* **17** no. 12 (2011), 2528–2535. https://doi.org/10.1109/TVCG.2011.205.

[143] Y. Wang and W. Peng, Interactive metro map editing, *IEEE Trans. Vis. Comput. Graph.* **22** no. 2 (2016), 1115–1126. https://doi.org/10.1109/TVCG.2015.2430290.

[144] J. M. Ware, G. E. Taylor, S. Anand, and N. Thomas, Automated production of schematic maps for mobile applications, *Trans. GIS* **10** no. 1 (2006), 25–42. https://doi.org/10.1111/j.1467-9671.2006.00242.x.

[145] H. Wei, Y. Wang, G. Forman, Y. Zhu, and H. Guan, Fast Viterbi map matching with tunable weight functions, in *SIGSPATIAL 2012 International Conference on Advances in Geographic Information Systems, SIGSPATIAL'12, Redondo Beach, CA, USA, November 7-9, 2012*, ACM, 2012, pp. 613–616. https://doi.org/10.1145/2424321.2424430.

[146] C. Wenk, R. Salas, and D. Pfoser, Addressing the need for map-matching speed: Localizing globalb curve-matching algorithms, in *18th International Conference on Scientific and Statistical Database Management, SSDBM 2006, 3-5 July 2006, Vienna, Austria, Proceedings*, IEEE Computer Society, 2006, pp. 379–388. https://doi.org/10.1109/SSDBM.2006.11.

[147] H. Whitney, Congruent graphs and the connectivity of graphs, *American Journal of Mathematics* **54** no. 1 (1932), 150–168. https://doi.org/10.2307/2371086.

[148] W. Winkler, String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage, 1990.

[149] S. Winter, Modeling costs of turns in route planning, *GeoInformatica* **6** no. 4 (2002), 363–380. https://doi.org/10.1023/A:1020853410145.

[150] A. Wolff, The map-labeling bibliography, https://i11www.iti.kit.edu/~awolff/map-labeling/bibliography/, Accessed: 2022-02-17.

[151] H. Wu, B. Niedermann, S. Takahashi, M. J. Roberts, and M. Nöllenburg, A survey on transit map layout - from design, machine, and human perspectives, *Comput. Graph. Forum* **39** no. 3 (2020), 619–646. https://doi.org/10.1111/cgf.14030.

[152] H. Wu, S. Takahashi, D. Hirono, M. Arikawa, C. Lin, and H. Yen, Spatially efficient design of annotated metro maps, *Comput. Graph. Forum* **32** no. 3 (2013), 261–270. https://doi.org/10.1111/cgf.12113.

[153] H. Wu, S. Takahashi, C. Lin, and H. Yen, Travel-route-centered metro map layout and annotation, *Comput. Graph. Forum* **31** no. 3 (2012), 925–934. https://doi.org/10.1111/j.1467-8659.2012.03085.x.

[154] X. Xie, W. Liao, H. K. Aghajan, P. Veelaert, and W. Philips, Detecting road intersections from GPS traces using longest common subsequence algorithm, *ISPRS Int. J. Geo Inf.* **6** no. 1 (2017), 1. https://doi.org/10.3390/ijgi6010001.

[155] Z. Zeng, A. K. H. Tung, J. Wang, J. Feng, and L. Zhou, Comparing stars: On approximating graph edit distance, *Proc. VLDB Endow.* **2** no. 1 (2009), 25–36. https://doi.org/10.14778/1687627.1687631.

[156] K. Zheng, Y. Zheng, X. Xie, and X. Zhou, Reducing uncertainty of low-sampling-rate trajectories, in *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, IEEE Computer Society, 2012, pp. 1144–1155. https://doi.org/10.1109/ICDE.2012.42.