Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Finding candidate routes with intermediate stops for railroad scheduling on block systems

Marcel Richter

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Supervisor:	Heiko Geppert, M. Sc.
Commenced:	June 7, 2021
Completed:	December 7, 2021

Abstract

The high coordination effort involved in running a railroad network faces researchers with a collection of difficult planning problems. Among these problems are the routing problem and the scheduling problem in which train lines are mapped to one route / schedule respectively. Both problems serve to establish a conflict-free traffic plan, one by separating trains in space and the other in time. There are plenty of existing solutions for both problems already, however, they fail to maintain the efficiency to carry out planning for an extensive region like an entire country. An untapped opportunity for discovering new solution approaches is looking into other domains, as both problems share a lot of traits with similar planning problems from other domains. One such approach is the configuration-conflict graph based approach which Falk et al. [FGD+21] route and schedule computer network packets with. Because they combine routing and scheduling problem to be dealt with as a single problem, they rely on a sensible set of candidate configurations to withstand the exponential increase in configurations. Their adoption of a k-shortest path algorithm for picking candidate routes does not translate adequately into the railroad domain, because in highly detailed railroad networks the k-shortest paths are mostly identical, thus leaving their algorithm little spacial opportunities to avoid conflicts.

Our contribution in this thesis lies in supplementing algorithms like [FGD+21] which benefit from a small input set, with a sensible set of candidate routes, facilitating them to run efficiently. We propose two graph-based routing algorithms that find a set of alternative paths visiting a given list of stations (the train line) in order. One of our proposed algorithms, labeled the simple routing algorithm, implements Dijkstra in a fashion that is compatible with a railroad topology. In addition, it implements a framework to optimize for other goals than path shortness, mainly uniqueness of the various alternative paths in a result set. Alternatively, we propose the multi-dimensional routing algorithm which expands on the simple algorithm to find even more unique alternatives, at the cost of worse runtime scaling. Our evaluations show vast improvements in terms of the shared distance between the different paths of a result set compared to Yen's k-shortest path algorithm. The runtime evaluations show enhanced performance of the simple routing algorithm compared to Yen in the majority of scenarios. They also demonstrate that great thought needs to be given to the algorithm parameters, since all optimization goals need to be balanced against each other carefully.

Kurzfassung

Da der Betrieb eines Eisenbahnnetzes einen hohen Grad an Koordination erfordert, stehen Wissenschaftler vor einer Ansammlung an schwierigen Planungsproblemen. Unter diesen Problemen sind auch das Zug-Routenproblem und das Zug-Zeitplanproblem, in denen Zuglinien jeweils eine Route / ein Zeitplan zugewiesen wird. Beide Probleme dienen dazu einen konfliktfreien Verkehrsplan zu erzeugen, eines separiert die Züge räumlich und das andere zeitlich. Es gibt bereits einige Lösungen für beide Probleme, diese scheitern allerdings daran, die Effizienz beizubehalten um Pläne für weitläufige Regionen wie ganze Länder zu erstellen. Eine unausgeschöpfte Möglichkeit neue Lösungsansätze zu entdecken, ist es dabei einen Blick in andere Domänen zu werfen, da beide Probleme viele Gemeinsamkeiten mit ähnlichen Planungsproblemen anderer Domänen haben. Eine dieser Ansätze ist der konfigurations-konfliktgraphbasierte Ansatz mit dem Falk et al. [FGD+21] Routen und Zeitpläne für Datenpakete in einem Computernetzwerk planen. Da sie Routenfindung und Zeitplanung in ein einziges Problem verschmelzen, sind sie allerdings abhängig davon, eine sinnvolle Menge an Kandidatenpfaden zu besitzen um den exponentiellen Anstieg an Konfigurationen zu widerstehen. Ihre Verwendung eines k-kürzeste Pfade Algorithmus', um diese Kandidatenpfade zu wählen, lässt sich allerdings nicht gut in Eisenbahndomäne übernehmen, da in hochdetaillierten Eisenbahnnetzen die k-kürzesten Pfade größtenteils identisch sind und ihrem Algorithm so wenig Möglichkeiten geboten werden, räumlich einen Konflikt zu vermeiden.

Unser Beitrag in dieser Arbeit liegt darin, Algorithmen die von einer kleinen Eingabemenge profitieren, mit einer sinnvolle Menge an Kandidatenpfaden zu versorgen und es ihnen so zu ermöglichen, effizient ausgeführt zu werden. Wir präsentieren zwei graphbasierte Routenfindungsalgorithmen, die unter Eingabe einer geordneten Liste an Stationen (die Zuglinie), eine Menge an alternativen Wegen produzieren, die diese besuchen. Einer der präsentierten Algorithmen, den wir simple routing algorithm benennen, implementiert Dijkstra auf eine Weise die kompatibel mit der Topologie eines Eisenbahnnetzes ist. Zusätzlich implementiert er eine Umgebung die es erlaubt, für andere Ziele als Routenlänge zu optimieren, hauptsächlich die Einzigartigkeit der verschiedenen Alternativen. Alternativ präsentieren wir den multi-dimensional routing algorithm, der den simple routing algorithm erweitert, um noch differenziertere Alternativen zu finden, auf Kosten schlechterer Laufzeitskalierung. Unsere Evaluationen zeigen große Verbesserungen im Vergleich zu Yen's Algorithmus, bezüglich der Distanz die sich die Alternativen teilen. Die Laufzeitevaluationen zeigen größtenteils eine verbesserte Performanz des simple routing algorithm im Vergleich zu Yen. Sie zeigen auch, dass die mitgegebenen Parameter gut überlegt sein müssen, da eine sorgfältige Balance zwischen den verschieden Optimierungszielen erzielt werden muss.

Contents

List of Figures

1 Introduction

In 2017, for the first time ever, railroad traffic by train, metro or tram has overtaken bus & coach to become the second most important public transport method in the European Union [Mob21]. Unlike bus & coach transport, railroad traffic is seeing an increase in passenger kilometers year by year, only exceeded by the massive jump air traffic has made recently. Naturally, an increase in usage also comes with an increase in required effort to maintain said railroad traffic. Not only is it challenging to manage the large-scale railroad networks in large countries like the United States and Russia, but even small networks provide their own set of challenges. The high railroad density of smaller countries as Czechia and Germany [Uni] requires special attention paid to the coordination of trains. Historically, train drivers carried most of the coordination responsibilities, so high-level planning was a minor concern. In extreme cases they were even driving on plain sight, much like how automobile traffic works today. However, the numerous incidents in the past have proven this approach to be severly lacking. So, why does driving on sight work with automobile traffic but not with trains? The key problem lies in how little means a train driver has to prevent an imminent accident. Unlike a car, a train is confined to the path of the rails and can not evade by steering, so the only option of avoiding an accident is come to a halt in time. However, this is often not possible considering the breaking distance which is orders of magnitudes larger than the one of a car, due to high mass and low friction to the metal rails. So, responsibility had to be distributed to other entities which have a greater ability to prevent accidents before they become inevitable. One way responsibility was shifted away from the driver is the safety block system, which is in use practically worldwide today. With it, there naturally was also an increase in the amount of planning required to run a railroad system from the outside. This planning process has proven to not be a simple task, which is why numerous algorithms to solve parts of it more efficiently have been proposed, and still are today [LLER11] [ZKR+96] [SEF+21]. A part of why so many algorithms for the problem exist is that train planning is not one atomic problem, but a collection of problems. These range in abstraction, e.g. from planning where new railroad tracks should be built to real-time rescheduling around a single train running late. And, some of these sub-problems are not an entirely unique to the railroad domain, so it is reasonable to consider algorithmic solutions from other domains to be applied. One of these is the configuration-conflict graph based planning solution proposed by Falk et al. [FGD+21].

Falk et al. [FGD+21] provide an approach to solve the problem of planning schedules and routes, however, not for trains but for computer network packets. When trying to use this solution in the railroad domain, one particular problem becomes especially evident. Since Falk et al. [FGD+21] combine the issue of scheduling and routing into one problem, they need to shrink down the set of candidate routes to avoid having to work on a set of solutions with exploding size. In their original solution based in the networking domain, they pick candidate paths using a k-shortest path algorithm. In the railroad domain however, a k-shortest path algorithm delivers results with a lot of

room for improvement. The most prominent problem is that by nature of being the shortest paths, the paths picked in a highly detailed railroad-network differ only marginally and can barely be called alternatives in the real world.

The contribution of this thesis is to support algorithms like the one of Falk et al. [FGD+21] with algorithms that determine a sensible set of candidate paths. By using the algorithms proposed within this thesis, planning algorithms can benefit from the appropriate candidate set and improve efficiency and/or result quality. The contribution of this thesis is the following:

- We provide two penalty-based routing algorithms that, given a certain train line, calculate a sensible set of candidate paths. Both algorithms can be seen as alternative approaches to solving the patfinding problem. Our *simple routing algorithm* is designed to find results which are sufficient in most use-cases, while our *multi-dimensional routing algorithm* is designed to find optimal results, no matter the cost in efficiency. Both algorithms are parameterized to let the user define a personal tradeoff between travel time, changes of tracks and edge duplication.
- We provide a reference implementation of the routing algorithms in Python to showcase their workings on a real data set.
- We extended the existing *railroad-parser* [PGRR21] to provide a tool for parsing an OpenStreetMap data set into a graph that is well fit for block based routing. The generated graph is in a format compatible with the proposed routing algorithms and complements the graph with additional information used to improve the performance of routing algorithms.
- We evaluate the algorithms against each other, and the referential Yen's k-shortest path algorithm [Yen71]. The evaluations are also designed to inform the user about the strengths and weaknesses of the impact of the algorithms' parameters.

Our evaluations have shown that the proposed algorithms improve substantially over the reference algorithm, Yen's algorithm, regarding our optimization objectives. The evaluations also show runtime performance benefits using our simple routing algorithm in the majority of scenarios. The multi-dimensional algorithm achieves a runtime performance on par with the simple algorithm in many scenarios, but experiences significant hits to runtime performance when calculating long routes without intermediate stops.

The structure of the thesis going forward is as follows. Chapter 2 summarizes the preliminaries and specifies problem statement. Chapter 3 presents the reader with an overview of the current state of research, mainly centered on conflict-graph based solutions related to the one of Falk et al. [FGD+21]. Chapter 4 introduces to the block safety system. The chapter serves as a bridge between real-world safety management and the scientific model. Chapter 5 focuses on railroad-parser, which is the tool we use to generate our data graph. Since railroad-parser is not an application original to this thesis the chapter explains the basic concepts of it first, followed by the many modifications that have been done in an effort to adjust it to our use-case. In chapter 6 we propose our routing algorithms. In summary, the chapter explains how we handle two challenges. For one, how routing algorithms can be made to respect a railroad topology. And also, how routing algorithms can be designed to produce result sets which meet our optimization criteria. Chapter 7 evaluates both routing solutions, particularly against Yen's k-shortest path algorithm. Finally, we conclude the thesis in chapter 8.

2 Preliminaries

A *graph* is a versatile data structure that is essential for understanding this thesis. In notation, a graph $G = (V, E)$ is a tuple of a vertex set $V = \{v_a, v_b, ...\}$ and an edge set $E \subseteq \{\{v_x, v_y\} \mid v_x, v_y \in V\}$. Vertices and edges are the core elements of a graph. *Vertices* represent a physical or logical object. Illustrations usually portray them as a circle. *Edges* are sets of two vertices describing the relation of two objects and are depicted as a line between vertices. One exemplary use-case of graphs is modeling a computer network. Computer nodes are represented as vertices in the graph, and wireless or wired connections between two computers are represented as edges. The obtained graph can then be used to simulate how data travels through the network.

When a vertex is connected to an edge, we say that vertex is *incident* to that edge. The *neighbors* of an individual vertex are all vertices which are linked to it by an edge $N : v_x \mapsto \{v_y \mid \{v_x, v_y\} \in E\}$. A *path* of length *n* from a *source* to a *target* is a list of vertices where neighboring elements in the list must neighbor each in the graph $(v_1, v_2, ..., v_{n+1})\ is\ path \Leftrightarrow \{v_1, v_2\}, \{v_2, v_3\}, ..., \{v_n, v_{n+1}\} \in E$.

Since graphs are a generic concept, they often are altered or extended to fit a use-case. In a *directed graph*, an edge is no longer a set but a tuple $E \subseteq V^2$ instead. Unlike in sets, the order of elements in a tuple matters $(v_x, v_y) \neq (v_y, v_x)$, which is why tuples allow us to model directed relations from a *source* to a *target*. The definition of a neighbor is semantically unchanged, but formally altered to $N : v_x \mapsto \{v_y \mid (v_x, v_y) \in E \vee (v_y, v_x) \in E\}$. Directed graphs also introduce two additional functions. One is the notion of *successors* of a vertex v_x in a directed graph, meaning all vertices v_y, which have a directed edge with target v_x, giving us $Succ : v_x \mapsto \{v_y \mid (v_x, v_y) \in E\}$. We define the *predecessor* function analogous $Pred : v_x \mapsto \{v_y \mid (v_y, v_x) \in E\}$. The requirements of a path also change for a directed graph. A successor of any vertex element in the list now also has to be a successor of that vertex in the graph.

The second extension presented are *weighted graphs* $G = (V, E, W)$, which contain a weight function $W : E \to \mathbb{R}$. The weight function is a feature commonly used in routing algorithms. In a weighted graph, the length of a path is no longer determined by the number of edges it uses. Instead, a path's length is the sum of the weights of all edges traveled along. Goal of a routing algorithm is to identify a path minimizing or maximizing its length. In the thesis, we will define custom *attributes* in a similar fashion to a weight function. These custom attributes map either vertices or edges to custom attributes. Examples for this are the attribute functions lat and $long$, which map a vertex to a geographic latitude or longitude respectively.

2.1 Problem Statement

The train routing problem describes an issue where a number of different lines have to be assigned exactly one route each while avoiding any conflicts. Together with the train scheduling problem they allow for the creation of a timetable, which is an essential part of creating a railroad traffic plan

that enables traffic to run safely and efficiently. Existing routing problem algorithms struggle to determine results for large regions because of a lack in efficiency. For a subset of those, the use of an unfit algorithm to calculate a set of candidate routes, e.g. k-shortest path, is a key factor in their inefficiency. The main problem is that the k-shortest paths share a lot of common edges and therefore do not present good choices, since a conflict in an edge can only be avoided by a path that does not contain that edge. Additionally, many algorithms only allow to optimize for path shortness and neglect practicality attributes like changes of track. These problems cause deficiencies in quality of results or efficiency for routing problem algorithms.

This thesis tackles the problem of calculating a sensible set of candidate routes to be used by a dedicated routing problem algorithm. *Given an ordered list of stations calculate a set of alternative routes which optimizes for following attributes.*

1. The routes do not take unorthodox paths
2. The routes differ from each other substantially
3. The travel duration of the routes is low
4. The routes do not change rail tracks frequently

1. ensures practicality in the real world by excluding routes which are physically impossible or impractical to follow. Since we are working on a graph where theoretically any edge could be used, it is important for us to limit in that regard. Routes are *not allowed to turn around* at anything besides a buffer stop or station, *not* allowed to travel through a single or double slip *switch in the acute angle* and through a *crossing anything but diagonally*.

2. ensures that the alternative routes in the calculation result are actually useful to another algorithm by providing it more options to avoid conflicts. We measure the distinctness of resulting paths using two metrics. Shared distance metric s_1 given by Equation (2.1), compares the edges of a path with all other paths and counts its distance as shared if it is not unique.

$$s_1 = \frac{\sum_{p \in paths} \sum_{e \in p} W(e) * f_{e,p}}{\sum_{p \in paths} \sum_{e \in p} W(e)} \tag{2.1}$$

Equation (2.1) divides the sum of shared distance by the sum of total distance to receive the relative shared distance with codomain $0 < s_1 \leq 1$. The function $f(e, p)$ (Equation (2.2)) serves as an indicator function checking if edge e has been used in any other path.

$$f(e,p) = \begin{cases} 1 & \text{if } \exists p' \in paths : p \neq p' \wedge e \in p \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

Shared distance metric s_2 uses a different approach to measure path uniqueness by taking into account which other path the duplicate edge is part of. The idea is, that we compare edges on a path to path basis, instead of comparing one path with all other alternatives at once.

$$s_2 = \frac{\sum_{p \in \text{paths}} max(\{g(p,p') : p' \in paths \wedge p \neq p'\})}{|paths|} \tag{2.3}$$

$g(p, p')$ (Equation (2.4)) measures the similarity between two specific paths. Equation (2.3) then picks the maximum of that similarity function, so only the single path with the most shared distance is considered in this metric.

$$g(p, p') = \frac{\sum_{e \in p} W(e) * \mathbf{1}_{p'}(e)}{\sum_{e \in p} W(e)} \tag{2.4}$$

$\mathbf{1}_{p'}(e)$ denotes the indicator function for element e in set p'.

3. , our goal of low travel time, ensures passengers enjoy quick traveling opportunities and helps keep operational costs down.

4. ensures another aspect to route practically, which is track changes. Infrequent track changes help avoid repeated acceleration and deceleration and keeps passenger comfort high. It also helps with determining more realistic routes, by preventing the train from constantly switching between opposite tracks to get close to beeline.

3 Related Work

Planning railroad traffic has been established to be a complicated process. Hence, a great amount of research has been done on it, and still is being done today.

Lusby et al. [LLER11] authored a review article issued in 2011, which divides railroad traffic planning into seven different research areas: 1) Network Planning 2) Line Planning 3) Timetable Generation 4) Train Routing 5) Rolling Stock Schedules 6) Crew Schedules 7) Real Time Management. This thesis contributes to the area of train routing, which covers the problem of mapping a set of train lines to one suitable railroad path each. Besides ensuring path quality requirements like shortness, an algorithm solving the train routing problem has to take special care to avoid conflicts between different lines. The contribution of this thesis are algorithms to generate a set of candidate paths for a dedicated routing problem algorithm to pick from. It is written as part of an ongoing effort to adapt a recently proposed computer network traffic planning approach [FGD+21] to be used in the railroad domain. Falk et al. [FGD+21] built upon their earlier approach [FDR20] of using configuration-conflict graphs for scheduling and routing time-triggered packets that travel throughout a network. In particular, they plan traffic for time-sensitive networking [SCO18] which is built to handle real-time traffic, profiting of a comparatively homogeneous network of computers. The data type they use, conflict graphs, are a special type of graph used for finding valid combinations of value bundles, which are also known as configurations. In a conflict graph, vertices represent a configuration and edges represent the incompatibility of two configurations with each other. Falk et al. [FGD+21] model network traffic using packet flows, which flow from a source to a destination. First, the approach generates configurations on a per flow basis, which may vary in time-shift when the packets are sent, and the path taken by the packets. During this process, edges are drawn between any two configurations that are incompatible with one another. An exemplary conflict is a combination where two packets are supposed to be sent by one computer node via the same network link simultaneously. This is disallowed because Falk et al. [FGD+21] assume zero-queuing for the network. The second step is to extract valid scheduling solutions from the conflict graph. This problem resembles the maximum independent vertex set problem, also known as node packing problem [Edm65]. Goal of the problem is to find the largest vertex set, which does not have an edge between any of its elements. If this largest set contains a configuration for each flow, the set completely solves the planning problem. Otherwise, the algorithm jumps back to step one and creates additional configurations to solve the problem. The maximum independent vertex set problem is notorious for being hard to solve, as it is known that no algorithmic solutions to it may have a constant factor runtime unless $P = NP$ [Her06]. Hence, they greatly benefit from limitations to the configuration set, which is what this thesis tackles.

The use of conflict graphs to solve the train routing problem is certainly not unheard of. In fact, conflict graphs are used in most approaches to solve the problem [LLER11]. Falk et al. [FGD+21] distinguish themselves by proposing a heuristic for solving the maximum independent set problem more efficiently. It is called Greedy Flow Heap Heuristic, and works by greedily scoring and picking possible configurations for the independent set result. Even when using this heuristic, choosing a

sensible configuration set is important for achieving reasonable performance results. The paper uses Yen's k-shortest path algorithm [Yen71] to generate a set of candidate paths, which is ill-suited for railroad scenarios. The algorithm finds a set of shortest routes by iteratively 1) running a routing algorithm on the graph 2) deleting edges visited by the calculated path. By nature of finding the k-*shortest* paths, the paths determined by Yen's algorithm differ only in minor details. This is not desirable for building a small candidate set, which profits from alternative routes with substantial differences. That is because having large deviations holds a high chance that one alternative does not conflict. In conclusion, Yen's algorithm works decently on a graph with a high abstraction level, but becomes more ill-suited the more detailed a graph gets. Another standout feature proposed by Falk et al. [FGD+21] is a way to dynamically generate conflict graphs and their solutions. This enables one to add new flows at any time with little to no impact on network functionality. In practice, that could be the foundation for using the approach for real-time plan management, but we do not explore this potential within this thesis.

The level of attention paid to the set of candidate paths varies for existing solutions to the train routing problem. Many solutions do not mention specifics on how the paths are generated, thus we assume that, by default, all possible paths serve as candidate paths [Bur05] [CD07]. For most of those solutions, this is due to choosing a model which abstracts from detailed routes on a switch to switch basis to a point where routing is (almost) trivial. Hence, we do not consider them relevant as context for this thesis. There are two notable exceptions to this, that directly or indirectly utilize conflict graphs to solve the train routing problem and do not specify a method for picking candidate paths. Caimi et al. [CBH05] do not abstract, but narrow their data set to Bern and its immediate surroundings. Another model proposed by Caprara et al. [CGT07] abstracts, although not to a degree where determining paths is trivial, and they limit their examples to certain cities in Italy. Both achieve tolerable performance results by handling a smaller region and must be adjusted or extended to handle an entire country. When looking for recent research on the train routing problem, one can observe that the research of conflict-graphs in relation to solve the problem has slowed down a lot. After Zwaneveld et al. [ZKR+96] popularized the usage of conflict graphs in the late 90s and early 2000s, this popularity has died down quite a bit. Most of the recent research explores alternative approaches, for example, the use of construction graphs [SPD+16], constraints [ZT16], artificial intelligence [SEF+21] and many more. In conclusion, given the apparent difficulty of the problem, it is reasonable to say that no conflict graph based algorithm working on an unlimited path data set is usable without limitation to smaller regions or abstracting.

Some solutions use k-shortest path algorithms akin to Yen's algorithm. Those suffer from the same similarity problem as described previously, meaning they perform well with small graphs or if performance is a minor concern. Yin et al. [YHlL11] demonstrate their k-shortest path approach using Beijing's metro network, which is also abstracted to stations and their connections, so performance plays a small role. Riezebos and Van Wezel [RV09] is of special interest as it is yet the only paper to solely focus on making a candidate set. They also opted for a k-shortest path routing algorithm, although they offer additional features compared to Yen's algorithm. Being supported by the Netherlands Railways, practicality plays a larger role in their research. They reference a study on practicality that concludes human input to be essential for generating a practical solution. That is why they built their routing algorithms with the ability in mind to require or ban a route from using a specific track. Unless a significant amount of manual input is done when routing using their approach, aforementioned limitations of k-shortest path algorithms will apply.

Zwaneveld et al. [ZKR+96] and Herrmann [Her06] reduce the candidate path set using the conflict graph they build to solve the train routing problem. They remove all dominated paths, i.e. paths where the set of conflicts is a superset of another path's set of conflicts, thus being inferior in terms of compatibility. A shortcoming of this procedure is that the nonadjustable amount of different alternatives per line is quite high. Zwaneveld et al. [ZKR+96] demonstrate the approach's effectiveness using a case study on Zwolle. They report that 65% of their configurations were removed using this procedure, which may not be enough given that this still leaves them with 1,100 configurations. When focusing on a single station, this may be adequate, but on longer distance lines, the number of eligible paths scales super-linearly. Caimi et al. [CBH+09] attempt to mitigate this flaw in their own routing solution. They deem the procedure of Zwaneveld et al. [ZKR+96] too computationally expensive and see potential in taking the railroad topology into account. Their proposal is that candidate paths should not be removed after creating all configurations, but during the creation process instead. To do so efficiently, they divide the railroad network into many switch regions, within each of which they perform their routing. To limit the set of candidate paths throughout a switch region, they first eliminate all dominated paths. Besides the requirement of having a superset of conflicts, dominated paths also need to have the same entry and exit point, as they are not comparable otherwise. In a second step they remove remaining paths which are similar to each other. To do so, a target number of paths is specified for each combination of entry and exit point. The set of paths chosen to remain are those paths which together have the least amount of conflicts with routes of all other entry/exit points. We were discouraged from using this approach by two issues. While parametrising the number of chosen paths is a sensible addition, its effectiveness is not demonstrated, as the authors chose to statically set it to 1 in their evaluation. They state that further testing with the parameter should be done, but did not mention it in their succeeding publications. Also, Caimi et al. [CBH+09] do not provide an automated procedure for partitioning a region into switch regions, which is important when working on larger scale regions. Additionally, both approaches suffer from the flaw that the length or driving time is not taken into account when sorting out ineligible paths, only their compatibility. We saw a possibility to improve path utility by giving the user the ability to weigh up distance and driving time against compatibility. Given above reasoning, we concluded the algorithm is not a perfect fit for our use-case.

To end off the chapter we want to point out that even though we wrote this thesis to be used by an adaption of Falk et al. [FGD+21], its use is not limited to that. Instead, any algorithm profiting from a small set of candidate paths may benefit from using herein proposed algorithms. Besides that, configuration parameters are offered to achieve the flexibility to fit into different scenarios.

4 Modern train safety systems

This chapter elaborates on what safety mechanisms are in use today, how they are enforced, as well as which authorities and organizations are involved. Knowledge of the real world is important to create a faithful model that ensures real-world applicability of the algorithms proposed in this thesis. While they vary in a lot of details from region to region, most safety systems follow the same fundamentals by using *separation by fixed distance* [Pac21]. The exact implementation of the separation by fixed distance approach varies by country. However, carrying out research without having a clear definition of a concrete safety system bears a high danger of over-abstraction. So, while our model is compatible with most regions, we settled on using Germany as an example for this thesis, primarily because of its high railroad density and extensive OpenStreetMap coverage.

We divide this chapter into several sections. Section 4.1 explains fixed distance separation and why the safety procedure prevails to be a standard used all around the world. Section 4.2 explains how train safety authorities can enforce fixed distance separation. Section 4.3 points out where fixed distance is anchored in Germany's train regulations.

4.1 Spatial distance separation procedures

Train safety systems strive to provide excellent safety but also maximize the network utilization. A good approach for finding a fitting safety system is therefore to start by evaluating systems which block a small amount of railroad tracks and thereby have a low impact on utilization. Then, find flaws in the evaluated system and propose a new one which fixes these flaws at the expense of more blocked track length. This way excessive blockage limiting train throughput, i.e. the number of trains passing through a rail in fixed time, can be avoided.

In an ideal scenario, *relative stopping distance separation* [Pac21] can be used to secure train rides. In this model, trains are secured by separating them with a minimum headway, that is analogous to the difference in their stopping distance. The stopping distance of a train comprises the reaction distance and the breaking distance. Reaction distance is a term denoting the distance a train travels until its operator has engaged its brakes $reaction = (time_{reacted} + time_{engaged}) * velocity$ and encompasses the reaction time and the time to engage the rail vehicle's brakes. The second part of the stopping distance is a train's breaking distance. Two trains may not share the same breaking distance if they are of a different model, use a different amount of wagons, carry differently weighted loads, or travel at different velocities. The breaking distances of trains A and B are noted as $breaking_A$ and $breaking_B$. The headway a trailing train B needs to keep to a leading train A can be calculated as $min\_headway = max(breaking_B - breaking_A + reaction_B, 0)$. In this model, train safety is achieved using a wandering hazard point, which marks the end of train A. A problem with the model is that it does not provide a way of modeling static hazards, such as

unlocked switches capable of derailing a train. Additionally, it does not assure safety if the leading train has an accident in which it slows down faster than its stopping distance. Both make the model unfit to be used in a real-world scenario.

The *absolute stopping distance separation* [Pac21] model is an alteration that fixes both these problems. In this procedure, every hazard point is treated as static, i.e. does not have movement information attached. This enables us to model stationary hazards, like unlocked switches, as well as avoid an instantly stopping train. Note that the hazard point of a leading train is still wandering as the train moves, but at any particular point in time, we treat it as standing. So, B no longer needs to consider the stopping distance of A, but only its own. Thus, the headway formula can be simplified to $min\_headway = breaking_B + reaction_B$. When in motion, a train has to determine which hazard point in his front is closest and keep minimum headway to this exact point. This is not trivial as the hazard point looked upon will keep changing as hazard points move, are added to model, or removed from it. As an example, consider a scenario where the train in front passes a switch ought to be re-positioned. Before the leading train passes the switch, the train is the closest hazard point, while the safely locked switch is not represented at all. After the train passes, signaling control unlocks the switch and starts changing its position. During this transition process, the switch is now the closest hazard point. As soon as the switch has completely changed its position and is locked, it is no longer a hazard point. Hence, headway needs to be maintained relative to another leading train or unlocked switch again. The reason this approach is not used to secure train safety is technological. Absolute stopping distance separation dynamically evaluates the required distance while both trains are in motion. For this system to work, every train requires on-board components continuously sending their position. Additionally, an on-board decoupling check component needs to be present to ensure no wagons are lost during movement. Else, in case of a decoupled wagon, this wagon presents a new static hazard point, which isn't registered in the model, since it does not have its own position tracker. A decoupling check component like that does not exist for freight trains, which is why this method is not practically used.

Fixed distance separation [Pac21] is the most commonly used system for securing trains worldwide. In this final alteration, distance is no longer evaluated from the standpoint of a train but instead of from the rail track. Rails are divided into *safety blocks* with a fixed length $length_{Block}$, usually 1 up to a few kilometers. Trains are only observed at the entry and exit point of a block. This enables us to avoid the decoupling problem of the previous model by counting the number of wagons entering and exiting a block at these two points. A block is inaccessible until the exit point counts that as many wagons exited as have entered the block before. Another advantage of fixed distance separation is does not require sophisticated technology to work. Since communication of safety information only needs to be done between the static entry and exit point, not with the mobile train, it could be implemented without the need for wireless communication. Despite this simplicity, given the system is correctly implemented, it is guaranteed that two unrestricted trains travel with a headway of $length_{Block}$. But this only applies to unrestricted trains, the distance between trains restricted by a blocking signal can potentially shrink all the way to zero. If a train stops right at the beginning of a new block, the following trains can still drive up to that beginning which brings them dangerously close together. Hence, implementation details explored in the following section are important to actually make this system safe.

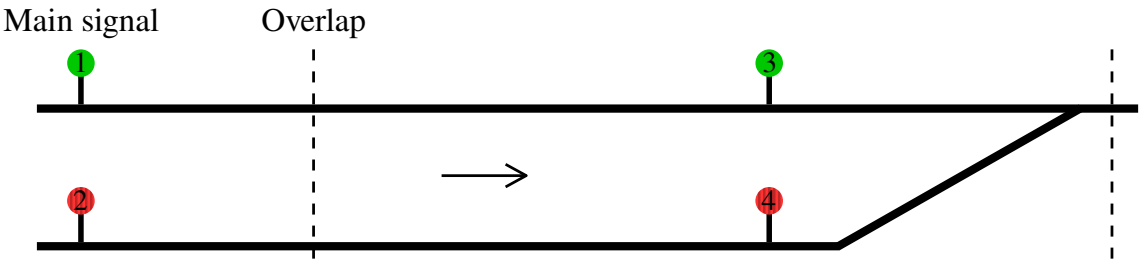
4.2 Enforcing spatial distance separation

Most real-world implementations of fixed distance separation use stationary signals to mark the borders of a safety block.rt Usually, these signals work using one or more lights where different colors and patterns have different meanings. Alternatively, there are semaphore signals, which historically were the favored signaling installation. Unlike light signals, these signals feature movable parts. Most prominent are signals with up to a few rotating bars, which train operators interpret based on rotation. Most feature additional lights to improve distinctness during nighttime. Even though new installations of semaphore signals have stopped a long time ago, they can still be found in use to date. How the determination of block entry and exit of a train are technologically implemented varies [Pac21]. Most solutions have in common that they transfer an electric or optic signal upon entry and exit to a train oversight facility. Some solutions are semi automatic, meaning they can determine that a train enters and exits automatically, but require manual input from signaling control to confirm the end of train marker has passed the exit. Otherwise, there is a possibility that the train has lost a wagon inside the safety block. Modern solutions are fully automatic, so they can guarantee that no wagon of the train remains, for example, by counting axles.

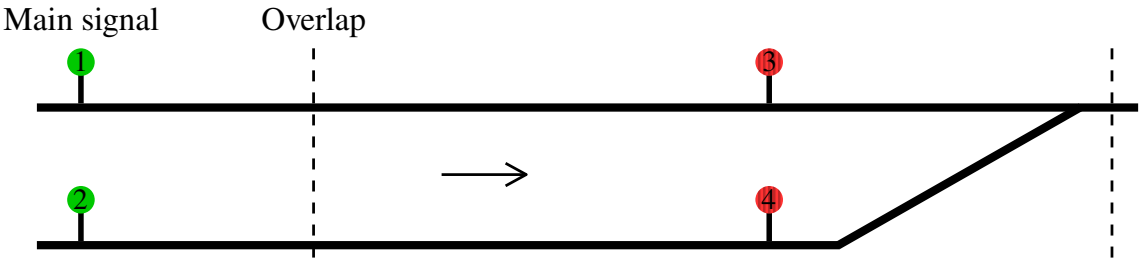
The *main signals* marking the entry of a new safety block alone are not enough to secure actual train traffic. If relying on sight sufficed to come to a halt before a main signal, sophisticated train securing systems would not be needed in the first place. If a train could easily stop before a signal just by sight, it could also stop in front of hazard points by sight. The solution to not relying just on sight is *distant signals*. They give the train operator an expectation of what the upcoming main signal shows. When the operator knows that the next main signal they pass is going to signal halt, they can slow down the train's traveling speed to a level where they can comfortably break before the stop signal.

If the safety section covered by a signal contains crossings, switches or buffer stops, it is technically no longer a safety block. Instead, the *route protection system* is used [Pac21]. It determines whether a section entry signal may be passed on a per route basis. To do so, the current switch positions are evaluated to find all possible routes throughout the section, i.e. until the next main signal. While all switches are locked, all routes can be determined before clearing the section for driving. The routes are then examined for conflicts where routes intersect at any point of the track. If two routes are in conflict with each other, thus mutually exclusive, the system has to decide which of these routes will be clear to travel through. The evaluation of clear tracks needs to be done whenever the state of a switch changes. Going forward, we will use the term *safety section* to encapsulate safety blocks as well as track sections covered by the route protection system.

Up until now, we always assumed that the control length of a signal is the same as the length of a block. However, many countries prefer control lengths longer than their block lengths. Doing that results in an overlap of two sections [Pac21]. For example, Germany primarily has block lengths of 1000 -1300 meters [Eis20b] plus 200 - 300 meters of overlap for each block, depending e.g. on the track's slope [Pac21]. The main purpose of having overlap is to account for human error. A security report by German *Eisenbahn-Bundesamt* for incidents in 2020 [Eis20a] counts 542 cases of operators failing to stop at blocking main signals. This amounts to one incident every 511 thousand kilometers driven on rail. In 455 of those incidents, the operator managed to stop their train without leaving the last signal's control length, i.e. inside the overlap. This poses a low risk to safety, since the security system guarantees safety in this case. It takes two trains overrunning



(a) Signal arrangement when the upper train does not stop



(b) Signal arrangement when the upper train stops

Figure 4.1: Handling overlap containing merging tracks. A green signal means a clear track, a red signal means stop.

at the same time, to pose a high risk to safety. This is illustrated in Figure 4.1. The figure shows two different scenarios, in each an upper and a lower train drive in parallel towards a merging track section. Figure 4.1a shows how intersections in the overlap are handled by the safety system. The upper train is given priority in the merged part of the tracks and may pass through the whole section. Meanwhile, the lower train has to stop at ② already, even though signaling stop at ④ would seemingly be enough. A signal covers the distance until the end of the section, as well as the following overlap. So, since the merging section inside the overlap is already assigned to be used by the upper train ② has to signal stop. If the signal did not cover the overlap, i.e. ② signaled green, safety when over running stop signal ④ could no longer be guaranteed. Figure 4.1b shows another case in which neither of the trains enters the merging section. Because the overlap is supposedly clear, both ① and ② can signal a clear track and trains can drive up to ③ and ④. But, if both trains overrun the stop signal into the overlap, a collision may happen at the intersection. In contrast, if only one of them fails to break and stops within the overlap, safety is guaranteed as that is covered by ① or ②, respectively.

4.3 Railroad economy and regulations in Germany

The following section summarizes important institutions and regulations that shape the German railroad economy. Its purpose is twofold. Firstly, to present the reader with a background of the foundation of previously explained safety mechanisms in legislation. Secondly, it explains and provides context on where in the economy our solution is applicable.

In Germany, railroad business is separated into two distinct types of corporations, railroad infrastructure companies and railroad traffic companies [Gov21a]. The most important company in Germany is *Deutsche Bahn AG*, which is in a unique position in the railroad industry. Although lawfully being a private company owned by shareholders, it is effectively state-owned, since the government owns 100% of shares and none are up for sale. The company owns over 600 subsidiaries with a mix of infrastructure companies and traffic companies.

Infrastructure providers build and maintain railroad tracks as well as additional hardware required to operate the track such as rail signals and signal boxes [Gov21a]. The dominant provider of railroad infrastructure in Germany is *Deutsche Bahn*'s daughter corporation *DB Netz AG*, which provided 33,286 kilometer of railroad routes in 2020 [Deu21]. This is a big margin over competitor and second biggest infrastructure provider *Deutsche Regionaleisenbahn Gruppe*, that provided 367 kilometer of railroad to its clients [Deu20]. Despite *DB Netz AG*'s lead in size, there is a large variety of 176 companies that were licensed to offer railroads to their clients in 2017 [Eis]. Most of them are companies owning small parts of railroad track in the local region. Many are combined companies which do not only manage infrastructure but also operate trains on it on their own. Infrastructure providers also manage the coordination of trains on their infrastructure [DB]. For said coordination, the provider maintains a schedule of all planned train lines and makes sure no conflicts arise. They also plan the exact path of a train throughout a line, i.e. on which track a train drives, but not train lines on a level of visited stations. The infrastructure provider staffs signal boxes which coordinate running trains [Deu]. This means that infrastructure providers are liable to comply with government issued train regulations, such as the AEG [Gov21a], and carry the responsibility of avoiding safety hazards during train rides. The infrastructure providers' business model is to offer traffic companies a license to operate a train on their infrastructure, billing them in kilometers driven on their tracks [DB]. Infrastructure providers are required by law [Gov21b], to provide a terms of service document which mandates the traffic company to adhere to the providers' conduction. In addition *DB Netz AG* publishes a set of regulations for train operation, with the most notable example being the *Fahrdienstvorschrift* [DB 19]. The *Fahrdienstvorschrift* is a document written to guide *DB Netz AG* personnel on safe operation of safety provisions and also inform train operation employees on how to react to various signaling.

A traffic company's role is to manage an inventory of rolling stock, employ staff operating the vehicles, as well as plan schedules on a station to station level. The railroad traffic business is more varied than the infrastructure provision industry. In 2020, a total of 447 railroad traffic companies have been licensed to publicly provide railroad transportation services to customers, over 332 of which actively used this license [Bun21]. The most competitive economic segment in Germany is freight transport. Freight transport is the only market segment the market share of *Deutsche Bahn AG* is below half, with 49% recorded in 2018 [mof20]. Following up with 7% market share is *TX Logistik AG*, a subsidiary of Italian state-owned rail company *Ferrovie dello Stato Italiane*. 11 more company's follow with a market share over 1%, including partially state-owned share companies (*SBB Cargo International AG*), private companies (*Captrain Deutschland GmbH*) and private share companies (*Havelländische Eisenbahn AG*). Long-distance passenger services stand in stark contrast to the freight transport economy, as they struggle to compete with *Deutsche Bahn AG*. The sum of all *Deutsche Bahn AG*'s competitors' market share is at a comparatively low 4%, but a clear upwards trend can be seen here largely because of newcomer *Flixtrain GmbH* and extensions of Austrian *Österreichische Bundesbahnen* AG into Germany[Bun21]. The last segment of the railroad traffic economy is local rail transport, where following *Deutsche Bahn AG*'s market share of 65% are French *Transdev Deutschland GmbH* with 6.5% and Dutch *Abellio GmbH* with 5.5%.

5 Acquiring a railroad network graph

To demonstrate and evaluate our proposed routing algorithms, a suitable set of test data is essential. This chapter revolves around the extraction of our test data from OpenStreetMap, as well as the ensuing data refinement steps. Our principle when creating this data set is like following. 1) Start from realistic OpenStreetMap railroad data as a baseline 2) Correct data by removing implausible constructs 3) Complement data with generated data which is plausible, but not accurate to reality. Data is saved in form of a graph, as is common practice in the routing domain. This also serves to provide the best data set compatibility For this same purpose we maintain an objective of keeping the data set as general as possible. Every graph attribute, with the exception of *type*, should be an optional addition and not worsen routing results of other algorithms. Unfortunately, railroad network data could not directly be obtained in sought-after format, so part of this thesis was to extract test data ourselves. The extraction process is largely based on a tool called *railroad-parser*. It has been developed as part of a previous project on the extraction of use-case specific data from an OpenStreetMap dataset [PGRR21].

The first section summarizes relevant existing functions of railroad-parser. The second section lays out our modifications to make railroad-parser fit our use-case.

5.1 State of railroad-parser

Railroad-parser provides a lot of functions which we could build our application upon. It serves as a good starting point, already offering capabilities to turn OpenStreetMap data into a railroad graph and exporting it as an edge/adjacency list and a list of stations. Nonetheless, most features were adjusted to better fit our use-case. So, in order to understand the modifications done by us to railroad-parser it is important to address the state of railroad-parser upon starting this thesis.

5.1.1 Parsing OpenStreetMap data

OpenStreetMap, the data source used in railroad-parser, is an “initiative to create and provide free geographic data, such as street maps, to anyone” [Opea]. It’s closely tied to the OpenStreetMap Foundation, which is a private company limited by guarantee, stating they are “supporting, but not controlling, the OpenStreetMap Project’ [Opea]’. OpenStreetMap provides an extensive set of general purpose map data for all around the world, especially covering Europe and North-America the best. All of this data comes from user contribution, which brings the advantage that the licensing terms provide great usage liberties. However, like any project relying on numerous volunteer contributors, it has its limitation in terms of correctness, completeness and consistency.

OpenStreetMap data formally comes as a list of three types of entities, which are described in the following sections. All of these entities can be supplemented with meta information using *tags*. OpenStreetMap tags are saved as pairs of key and value. Theoretically, contributors could make up their own key and value formats as they please. However, contributors stick to conventions laid out in the Wiki [Opeb] to make their information more easily usable. In this convention keys and values follow a tree structure where children and parents are separated by a colon, e.g. *railway:signal:main= DE-ESO:ks*.

Nodes represent "a single point in space defined by its latitude, longitude and node id" [Opeb]. Practically, they are used for two purposes. First, to represent a thing that was, is, or will be physically present and which the spatial extent of is negligible. An example of such an object is a railroad signal, which the position of is much more relevant than its size. The second purpose of nodes is to be grouped to represent a larger data object like a way.

A *way* is "technically [...] an ordered list of nodes" [Opeb] and can represent a number of things. The most intuitive thing to represent with a way are things where only extent in one dimension is relevant. For example, a physical path, like a railroad path, is represented as a way. But, ways can also be used to model an object with two-dimensional extension, so basically an area. The requirement for making an area is to have a closed way, a way where start and end node are identical, thus forming a polygon. Based on the type of way, OpenStreetMap implementations automatically choose how to interpret a closed way. Alternatively, OpenStreetMap contributors may also manually dictate this decision using a tag.

The third and last data entity are *relations* which are "ordered list[s] of one or more nodes, ways and/or relations" [Opeb]. They allow contributors to define a more complex concept. A train related example for this are relations that define a scheduled train route, like a regional train hourly commuting between two cities. One can create the route in OpenStreetMap by creating a relation and adding all ways and station nodes visited as part of the route.

Railroad-parser serves as a tool which works on top of this dataset to perform two main functions. One is filtering for railroad traffic related entities of the OpenStreetMap data set, which contains all kinds of data, such as streets and buildings. The other function is to transform the filtered data into a graph. The filtering process is done using an existing framework for parsing OpenStreetMap data called PyOsmium [Sar]. To filter for railroad related entities, railroad-parser iterates over all ways and checks for a tag with key *railroad*. If a way does have a railroad tag with value *rail*, all of its nodes are added as vertices to the graph. Additionally, edges are drawn between any two neighboring nodes. While parsing the OpenStreetMap data, railroad-parser adds three new attributes to the graph. The graph attribute *type* is set, which maps vertices to their function $type : V \to \{buffer\_stop, signal, station, switch, crossing\}$. The geographic latitude and longitude are also saved in attributes $lat : V \to \mathbb{R}$ and $long : V \to \mathbb{R}$. Height data has not been included in this positional data, as it was not considered to influence distances to a noteworthy extent.

5.1.2 Calculating edge lengths

The goal of this step is to add an edge attribute $length : E \to \mathbb{R}$, which corresponds to the rail distance a train has to cover when driving between its incident vertices in the real world. Having a distance attribute is crucial for many use-cases. Notable examples are route planning and traveling

time calculation. The accuracy of this calculation is limited by the OpenStreetMap way's resolution. If a way contains densely packed nodes, it approximates its real curvature more precisely, much like the approximation of a circle with a polygon. So, to achieve results which are as precise as possible, distance calculation is performed before any nodes are removed or added. Railroad-parser determines an edge's length based on its incident vertices' geographical positions. Their latitude and longitude are used to determine the *geodesic*, the shortest curve connecting two points on the surface of an ellipsoid. Railroad-parser does not have an own implementation for this, but instead passes latitude and longitude of both vertices to GeoPy [geo], a library of functions to ease geographic calculations. GeoPy's implementation is based on a geodesic distance algorithm as written down by Karney [Kar13].

5.1.3 Determining switch directions

Another feature of railroad-parser is its algorithm for determining switch directions. In the graph, switches are portrayed as vertices with three neighbors, but in reality, we can not always interpret them like that. In a switch, a railroad diverges into two separate railroads, so a train may not travel from one part of the diverging side to the other. The information on which two of these neighbors belong to the diverging side is not included in OpenStreetMap tag information. Even so, this information can be calculated based on the provided geographic data. Let the investigated switch be v_{switch} and its neighbors $N(v_{switch}) = \{v_a, v_b, v_c\}$. From the graphs position attribute, we can calculate three vectors, each shifting from v_{switch} to one neighbor.

$$\vec{a} = \begin{pmatrix} lat(v_a) \\ long(v_a) \end{pmatrix} - \begin{pmatrix} lat(v_{switch}) \\ long(v_{switch}) \end{pmatrix} \tag{5.1}$$

$\vec{a}$ is calculated as shown in Equation (5.1), $\vec{b}$, $\vec{c}$ are calculated analog. The angle between these vectors then provides information on the switch directions.

$$\delta_{ab} = arccos\left(\frac{\vec{a} \bullet \vec{b}}{||\vec{a}|| * ||\vec{b}||}\right) \tag{5.2}$$

Three angles are required δ_{ab}, δ_{ac} and δ_{bc}, each is calculated by applying Equation (5.2). From these three angles, we determine the smallest one. The two vectors in the index are the two vertices that are on the diverging side, thus traveling between these two is not possible. This data is captured using two attributes $single\_side : V \to V$ and $diverging\_side : V \to \{X \mid X \subset V \wedge |X| = 2\}$.

5.1.4 Intermediate vertex removal

After calculating the switch direction, railroad-parser optionally removes intermediate vertices from the graph. Intermediate vertices are vertices with two neighbors, which aren't assigned a special function, unlike e.g. stations. A user may opt to remove these vertices to shrink their data set without any loss of information, as intermediate vertices do not present a choice for routing. The set of intermediate vertices consists of a number of different node types with no meaning to most calculations. Examples for included node types are milestone nodes, which represent milestone signs at the side of the railroad, or OpenStreetMap way points, which do not represent any real-world object, instead they are used to model the curvature of a railroad. To remove intermediate vertices,

railroad-parser iterates over all non-intermediate vertices. These vertices serve the purpose of a starting point from which railroad-parser starts the vertex removal. The algorithm moves along each neighbor until a non-intermediate vertex is found. All vertices between starting and ending point are removed and finally a new edge is drawn between start and ending vertex. The length of the new edge is set to the sum of lengths of all removed edges. This way, length calculation does not need to be repeated and maintains its original precision.

5.2 Adjustments made to as part of this thesis

While railroad-parser did already provide important features, such as OpenStreetMap parsing, it needed adjustments to deliver results that fit our use-case. After all, graphs generated by the application did not even contain signals, which are obviously essential for this thesis. Our adjustments range from simple additions, e.g. duplicate edge removal, to modifications affecting most of the application, e.g. directed graph usage. The following sections will go over all significant changes to the application as part of this thesis.

5.2.1 Making the graph directed

Most railroad tracks are usually driven on in one way, so in order to find realistic routes, our routing algorithm should respect directions. Directions were ignored by railroad-parser up until our changes, as it used an undirected graph as its data format. Now, the preferred directions get extracted from OpenStreetMap using way tag *railway:preferred_direction*. The field can have one of the three values *forward*, *backward* or *both*. Since OpenStreetMap ways are ordered lists of nodes, all of them have a direction by themselves, so *forward* and *backward* give the direction relative to the direction of the way. Not all railroad OpenStreetMap ways actually possess this tag, which is why a default value must be introduced. We chose this default value to be *both* for compatibility reasons. The directional edges are only used to model this conventional direction, not the actual logic of a switch or crossing element, which is done by the routing algorithm instead. All of railroad-parser's filtering algorithms had to be adjusted to support the new directed graph format. Especially noteworthy is the intermediate vertex removal algorithm. In scenarios where multiple edges are combined into one, it is possible that multiple edges with different directions have to be combined, leaving it unclear which direction the combination has. Once again, we chose a permissive approach to solve this problem by making these edges bi-directional.

5.2.2 Adding signal vertex type

Main signals vertices are crucial for our routing, as they indicate the start and end of a safety section. Before our changes, they were treated as intermediate vertices and therefore removed from the graph as part of the intermediate vertex removal. We modified the OpenStreetMap parsing algorithm to give the type *signal* to following main signal vertices:

H/V-Signals with tag *railroad:signal:main=DE-ESO:hp*. H/V is short for *Haupt-/Vorsignal (ger. main/distant signal)*, signifying its heavy reliance on discrete distant and main signals. Being introduced in the late 19th century, the H/V signaling system is deprecated, mostly to be replaced by Ks-Signals. H/V are the only block signaling system old enough to still feature semaphore signals. Despite being deprecated, H/V signals are the most used main signals in Germany today.

Ks-Signals with tags *railroad:signal:main=DE-ESO:ks* or *railroad:signal:combined=DE-ESO:ks*. Ks is short for *Kombinationssignal (ger. combination signal)*, it allows to combine distant and main signal into one. Being combined means having not only the ability to display stop or clear for its safety section, but also inform the train operator whether to expect a stop signal in the next safety section. Another characteristic of Ks signals is their simplicity having only 3 states.

Hl-Signals with tag *railroad:signal:main=DE-ESO:hl*. Before its reunification in 1990, post world war two Germany was seperated into the Federal Republic of Germany (West Germany) and the German Democratic Republic (East Germany). As Germany was divided, so was its railroad system. The *Deutsche Bundesbahn* was responsible for West Germany, while the *Deutsche Reichsbahn* provided railroad services to East Germany. Hl signals were first introduced to East Germany by the *Deutsche Reichsbahn* in 1959, serving as a combined signaling system intended to replace H/V signaling. In addition to all functionality of a Ks-Signal, they offered the ability to show certain speed limits, both for its block and for the next block.

Block signs with tag *railroad:signal:train_protection=DE-ESO:blockkennzeichen*. Block signs are traffic signs signaling the start of a block. They are used in a few major traffic lines, where no stationary signals have been placed, for example, following the new ETCS Level 2 standard [OUF18]. Instead of showing information on a signal, it is transmitted to the train via electromagnetic waves either from a transmission station (GPR-S for ETCS) or via conductor loops layed out alongside the middle of a rail (LZB).

These signals are combined into the new *signal* vertex type. This implies that they get their own color when plotting and are no longer removed in the intermediate vertex removal process. However, not all main signals do actually get this type. OpenStreetMap signals come with a tagged direction *railway:signal:direction*. For bidirectional ways, one could add a direction attribute for each signal and let a routing algorithm handle the logic. However, this dis-accords with our objective of making all attributes optional. If a routing algorithm did not evaluate this attribute, it would consider signals of both directions. Not only would such an algorithm see signals about twice of their actual amount, it also experiences large variation in their spacing. Often signals with two different directions are very close to one another, but this is not always the case. Considering this problem, on a bidirectional way, we only want to keep signals of one direction. As explained in the last section, if a way has a preferred direction tagged to it, we treat that way as being uni-directional. Thus, we only want to keep signals which face the preferred direction of the way. This is done using the *railway:signal:direction* tag.

5.2.3 Running intermediate node removal multiple times

The intermediate vertex removal algorithm was adjusted to launch multiple times. Because, even though the algorithm correctly removed all intermediate vertices, some vertices could still be found after the removal step. This is caused by following scenario. Assume we have two switches connected to each other on their divergent side, as in an *O* letter shape. Also assume, that on both

rails of the O, only intermediate vertices can be found, no signals or stations. If this scenario occurs, and it does many times in Germany, the intermediate removal algorithm does not produce a correct result. The algorithm will remove all vertices in between the two switches, making them connect directly. As the switches are connected by two different tracks, these tracks get reduced to two different edges during intermediate removal. This leaves us with a duplicate edge, which does not fit in our graph model and is removed from the graph later on. But even though the number of incident edges is preserved with this duplicate edge, none of the algorithms in railroad-parser handles vertices based on this number, but instead by the number of neighbors. The two switches considered now only have two neighbors and are thus recognized as intermediate vertices. The fact that the O gets by removed is not problematic by itself. After all, it does not contain any signals and therefore is a single safety section making overtakes impossible. What is problematic is that intermediate nodes are not allowed in our model, so a data set containing any is faulty. Launching the algorithm repeatedly, until no more change could be observed, is a simple solution which does fix the problem. One might think running it twice would be enough, but this is not the case since this scenario does occur recursively, meaning the O is on a left or right side of an even bigger O.

5.2.4 Adding a speed limit to edges

OpenStreetMap provides contributors with the tag *maxspeed* to specify a speed limit for tracks. This information is important to us, as our routing algorithms are travel time based. To calculate the time a train needs to travel between two vertices, not only do we need the edge's length but also the speed the train travels at. The $speed\_limit : E \to \mathbb{R}$ attribute is added while the edges are parsed from OpenStreetMap data. This happens after the edges direction is set, since the direction can influence the speed limit. If the direction is uni-directional tags *maxspeed:forward* and *maxspeed:backward* are used. The directionless *maxspeed* tag is used as a fallback or if the way is bi-directional. The speed limit specified on the edges is a track-sided speed limit. Speed may also be bound by the vehicle, but this information is given to the routing algorithm and not added to the graph. Special care has to be taken when averaging speeds, for example, during the combination of edges in the intermediate vertex removal algorithm. Contrary to first intuition, n speed limits can not be averaged by dividing their sum by n. In a sense, doing so averages the speeds over the track distance and not the speed of a vehicle traveling on it. This only gives correct results which maintain driving time if the time spent in each of the speed limits is the same. For proper averaging, the driving time should be evaluated first $time_{edge} = length(edge)/speed\_limit(edge)$. Then lengths and driving times are summed up and divided $\frac{length(edge1)+length(edge2)+\ ...}{time_{edge1}+time_{edge2}+\ ...}$.

5.2.5 Adding crossing vertex type

Crossings are vertices with four neighbors, where only traveling diagonally is possible. As such, they are a fixed railroad element and do not allow to navigate the train into different directions. Before, crossings were explicitly removed by railroad-parser and replaced by connecting the two different ways through the crossing directly. This was justified by the fact that, since crossings do not leave the train with a choice of path, removing them did not influence which vertices can be reached from where. However, crossings must be considered when looking out for conflicts between trains. Since a collision may occur if two trains travel through both of its diagonal ways, the crossing must be kept as a single node and may not be split into two different ways. Just as

done with switches, we must ensure that routes through crossings only take the single legal way through it. This is saved in an attribute $sorted\_neighbors : V \rightarrow V^4$, which allows us to reason on the crossings possible ways. This attribute is calculated in a similar way using provided geographic data. Let the crossing vertex be $v_{crossing}$ with neighbors $N(v_{crossing}) = \{v_a, v_b, v_c, v_d\}$. Using Equation (5.1) we can calculate the vectors $\vec{a}$, $\vec{b}$, $\vec{c}$ and $\vec{d}$. Three angles can be calculated δ_{ab} δ_{ac} and δ_{ad} as given by Equation (5.2). To ease evaluation of the angles, a normalization step is performed next. The normalization calculates the difference of an angle to 180 degrees, the value we expect a straight way through a crossing to have.

$$|\delta_{xy}| = \left\{ \begin{array}{ll} \pi - \delta_{xy}, & \text{for } \delta_{xy} < \pi \\ -(\pi - \delta_{xy}), & \text{else} \end{array} \right\} \tag{5.3}$$

From these three normalized angles, we determine the smallest one, giving us the counterpart vertex for v_a. Using that information we can conclude the two pairs of vertices which make up two diagonal path through the crossing. However, for the upcoming double slip switch removal we also need to know in which direction the path goes through the crossing. So, instead of having two partially ordered pairs, a complete order of all four neighbors has to be made. This vertex is determined by having the smallest un-normalized angle δ_{ay}.

5.2.6 Adding additional features to switches

Two different modifications have been done regarding switches. Each will be described in their own paragraph.

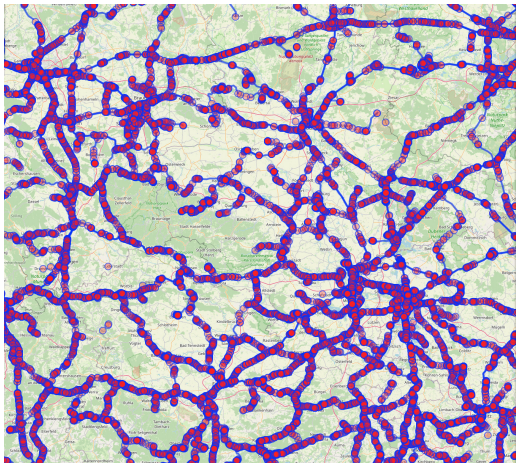
The original railroad-parser assumed a world where switches always have three entries/exits, two of which belong to a diverging side. It did not account for switches with four tracks, so called double slip switches. These switches are laid-out like crossings, but for each entry give two options on where the train can exit. Functionally, these work like two separate connected switches with diverging sides facing away from each other. In order to not further complicate the model with another vertex type, this is also what we convert them to. The algorithm removes the double slip vertex and replaces it with two single slip switch vertices. To determine in which way these vertices need to be connected, the $sorted\_neighbors$ attribute is calculated using the methodology of Section 5.2.5. Then, two new switch vertices are added to the graph and connected with an edge on their non-divergent side. One switch is then connected by edges with v_a and v_x, the other with v_y and v_z. Special care needs to be taken when setting lat and $long$ for the newly created algorithms. As to not mess up the calculation of the switch's direction, the position of the first switch is shifted from the double slip switch slightly towards v_a and v_x, and the second switch towards v_y and v_z.

The other addition regarding switches was an optional attribute called *default*: $V \rightarrow V$ which maps a switch vertex to the neighbor which is on the diverging side and the closest to just driving straight. This was introduced as a means to reduce changes in driving track on the calculated paths. While always switching to the track with least distance does achieve the path of shortest distance, in reality they slow a train down and also increase the chance of conflicts with oncoming traffic. Adding this attribute allows a routing algorithm to apply penalties to routes that do perform changes of track. The calculation step which sorts the neighbor of a crossing Section 5.2.5 already introduced the required equation. Using Equation (5.3) we can normalize the three angles δ_{ab}, δ_{ac} and δ_{bc} between the switch's neighbors so they reflect the difference to 180 degrees. The

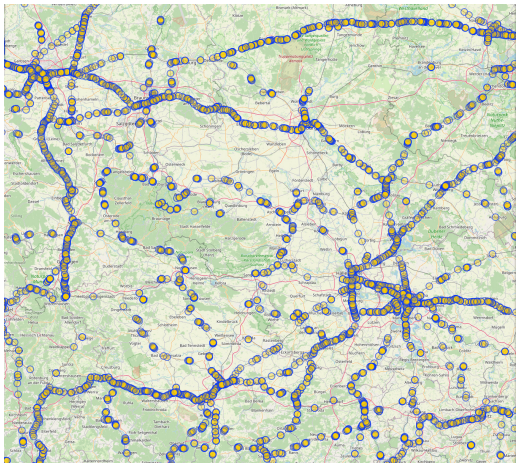
smallest normalized angle gives us the two vertices which form the straightest line through the switch. However, not in all cases do we actually set the *default* attribute. Not all switches fit into the switch pattern of having default and alternative exit. Many switches have a V-shaped divergent side, where neither side functions like a straight path. This includes the aforementioned double slip switches, which after being split up, usually look like two V-shaped switches. In order to prevent false penalties for this type of switch *default* is only set if a clear distinction between the two divergent paths exists. To determine whether the two paths are distinct enough, we look at the difference between their normalized angles. If they are distinct the threshold of 0.06 radiant difference is met and *default* is set. If the switch resembles a V-shape the difference is below the threshold and *default* is not set to avoid false positive penalties.

5.2.7 Refining the data set

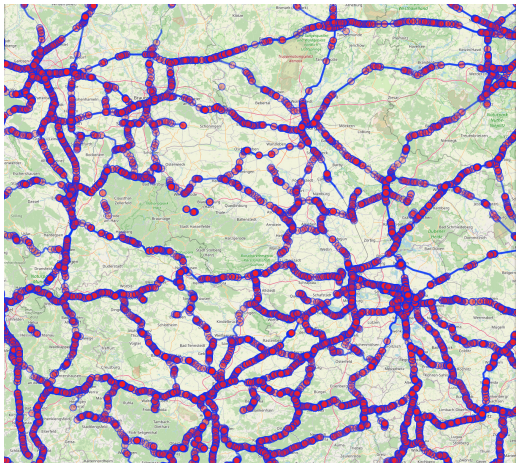
While OpenStreetMap does have its strengths like as its lenient licensing terms, it is not without problems. The biggest one is that, being reliant on user contribution for information, the data set it provides is not quite complete. While coarse geographic details like the railroad tracks are well captured, finer details like tag information often fall by the wayside. We illustrate this problem by visualizing the data in Figure 5.1. Figure 5.1a shall be seen as a reference and marks all active railroad tracks in an exemplary area of 50,000 km^2 in central Germany. Figure 5.1b highlights all main signals, regardless of direction, in the same area. The 5164 main signals which are tagged in OpenStreetMap do not cover the whole railroad network as they should. Since Germany uses block lengths of about 1.2 kilometers excluding overlap, we would expect a perfect data set to cover most of the track length with dense signals. We can visually see deficiencies both in coverage density, as well as in coverage area in the illustration. If we used this data set, which is unrepresentative of the real world, we would observe exceptionally large block sizes, which would hinder sensible scheduling in these areas. Hence, we developed a simple algorithm to insert artificial signals into ways where the signal density is not sufficient. The algorithm iterates over all edges and checks if $length(edge) > 2150$ meter. In such cases, the algorithm removes the edge, preserving metadata like length, and adds a new signal vertex to the graph. The vertex is then connected to the source of the removed edge by an edge of with a random distance uniformly distributed between 1000 - 1075 meters. The random length is then subtracted from the preserved edge length and if the remaining length is still over 2150 the previous step is repeated. We initially generated edges with a length of 1000 - 1200 meters, which is more in line with reality, but reduced the number to 1075 as some tracks were not covered when using larger numbers. That is because the algorithm looks at edge length, not distance between signals. So, since switches and stations divide a track of a safety section into multiple edges, long tracks without signals and frequent signals or switches will not be modified by the algorithm. In the end, 58.1% of final main signals were generated by this algorithm. Figure 5.1c shows the coverage of the speed limit tag for the investigated area. In numbers, 45% of the evaluated ways have a speed limit tagged. This number is a bit deceiving though, as it has a bias to make the coverage appear smaller than it is. The few large and significant ways on the main lines are more likely to be well tagged than the many small, insignificant branching ways. Still, one can visually identify differences between reference and Figure 5.1c mainly when it comes to side arms springing off the main lines. Despite the bias, we decided to set the speed limit of untagged ways to 92 which is the average speed limit accross Germany according to our OpenStreetMap data. Figure 5.1d illustrates the 10% of evaluated tracks that have a direction tagged. The poor coverage is the reason most of the edges in our resulting graphs are bi-directional. We were faced



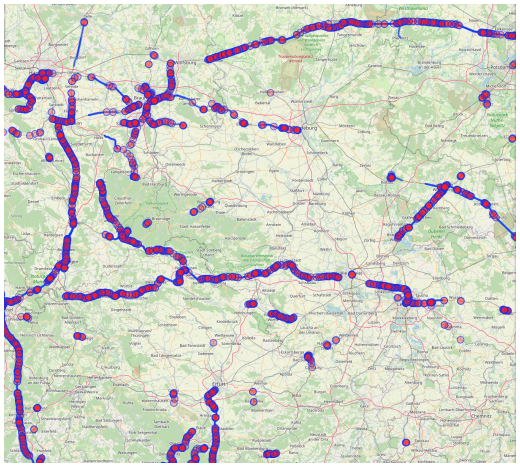
(a) All active railroad tracks



(b) Main signals



(c) Railroad tracks with tagged speed limit



(d) Railroad tracks with tagged direction

Figure 5.1: OpenStreetMap data completeness demonstrated on an area of 50,000 km^2 in central Germany. Data is provided and visualized by *Overpass Turbo* [Mar].

with two options to use as a default when no tag information is available, using the direction of the OpenStreetMap ways as a default or defaulting to bi-directional. Former option underapproximates the set of possible paths from what is actually possible, while the latter one overapproximates. The latter option was chosen to be used in railroad-parser. Not only is it more user-friendly than making a perfectly valid route evaluate to no possible paths, but it also serves as a better demonstration, since an algorithm picking alternatives profits a lot of alternatives to pick from.

We also encountered a notable amount of duplicated signal and station nodes. By that we mean two different nodes at varying positions describing the same signal or station. We verified the duplication cases by comparing the data set to public video footage and/or node metadata, such as the signal identification number. To fix the problem with duplicated signal and station nodes, we employed an algorithm to remove duplicated nodes. It iterates over all edges to check if their length is under 100 meters and the two incident vertices are both stations or signals. If that is the case, one of the vertices is removed and the edge is modified according, to maintain original metadata.

5.2.8 Smaller changes

Not all of our changes require enough explanations to warrant their own section. But, they still are important to get the full picture of what a graph generated using the extended *railroad-parser* looks like. This final section summarizes these small changes, each in their own paragraph.

We added an algorithm to convert self edges to buffer stops. Self edges may be created by the intermediate vertex removal algorithm. Source of the self edges are turnarounds, switches where both ends of the diverging side are connected to each other. If all nodes on the inside of this loop are intermediate, both ends of the switch become directly connected by an edge, forming a self edge. Few of these actually exist in Germany, nonetheless they should not be neglected. Functionally, they are equivalent to buffer stops, at least in our model. Both only allow the train to change directions. In reality, the difference is that the front remains in front when turning using a loop, but not when turning around a buffer stop. The conversion algorithm iterates over all self edges and removes them. Then a new buffer stop node is added alongside a bidirectional edge between stop and switch with the speed limit of the removed edge and length $length(selfedge)/2$. The length is halved to compensate for the fact that the new edge, unlike the original self edge, will be used twice when turning around using the buffer stop, once to enter the stop and once to exit.

We added an algorithm for removing duplicated edges. How duplicate edges are created is described in Section 5.2.3. The section also describes why the information lost as part of this process is not relevant. In the removal process, the edge with lowest temporal length prevails while all other duplicate edges are removed. The algorithm splits the list of all vertices into workload chunks to take advantage of symmetric multiprocessing.

We added export files to get all necessary information. In order to distinguish the different vertex types, one file for each type listing all of its members, is exported. The list for the switch type is expanded by the information contained in the switch direction algorithm. This allows us to plan realistic routes, where the train can not turn around in the middle of a track to drive between ends of the diverging side of a switch.

We corrected the OpenStreetMap dataset in two ocassians. OpenStreetMap can be edited by anyone possessing a free account and changes happen instantly without any approval required. In both occasions, the rail path in OpenStreetMap was not representative of the actual paths. One error featured a railroad station node, which was a crossing at the same time. To match the real-world layout, we split the station node in half and let the rails run each through one station in parallel. The other error featured a node with five neighbors in the middle of a rural multi-rail track. The construct had no base in reality and was replaced by us with new ways and switches to correctly portray the real routing.

6 Routing on the railroad graph

Having dealt with all prerequisites, we can focus on the actual routing algorithms proposed as part of this thesis. The purpose of the algorithms can be summarized as follows: *Given an ordered list of stations (a train line) calculate a set of paths visiting all of these stations in order.* The purpose of this set is to be used as candidate paths for a dedicated train routing problem algorithm[1]. To be useful as a candidate set, the algorithms are designed to optimize for four route qualities. The most important one is respecting the railroad topology and not taking any paths we deem illegal. For example, a train entering on the diverging side of the switch may only leave it on the non-diverging side, everything else would require the train to reverse. The second quality is alternative uniqueness. The more diverse the different alternatives for the lines are, the more options the train routing problem algorithm has to avoid conflicts. Third, we want to minimize the amount of track changes, as they slow a train down and increases chance of conflicts with an oncoming train. Lastly, we want to keep the travel time low. There are many reasons for this, customers expect quick travel times, long paths decrease the throughput of the railroad tracks, etc.

The following chapter explains how we generate the set of candidate paths on the railroad graph. We propose two alternative algorithms for finding a set of paths. Both algorithm are built on top of Dijkstra's algorithm and apply penalties to edges to achieve their optimization goal. The first algorithm, henceforth called *simple algorithm*, compares edges to a set of edges used in any previous path. The second algorithm, which we named *multi-dimensional algorithm* builds on top of that algorithm to provide better results for many alternatives, at the cost of higher complexity as well as run time and memory demands.

6.1 The simple routing algorithm

First of all, both algorithms share a common basic structure. Both algorithms are based on Dijkstra's routing algorithm, which is a keystone in routing research, even to this day. The algorithm can be implemented in various ways, so our implementation will be summarized. Our implementation is an iterative algorithm that maintains a min-heap with all the vertices it has discovered but not yet visited. The min-heap is sorted by the currently lowest known distance to the respective vertex, which is also saved in an array. To retrace the shortest path, each vertex remembers its predecessor on the path. To avoid loops, the algorithm maintains a set of all visited vertices and does not add those to the stack anymore. However, Dijkstra's algorithm needs adjustments in order to accurately model the rail topology, e.g. a switch. Without any restrictions, the algorithm can use any edge for its path including illegal moves like traveling across the diverging paths of a switch, or suddenly reversing. Our solution to this problem was to build a framework around Dijkstra which only

[1] The train routing problem should not be confused with the problem this chapter solves, which is pathfinding. The former problem is explored in more detail in chapter 3.

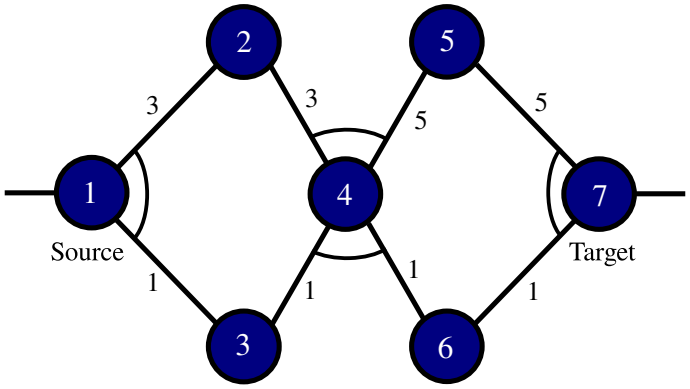


Figure 6.1: A simple counterexample to show that adding additional state to Dijkstra is not done easily.

exposes neighbors to it which we allow visiting. To be more precise, following restrictions are applied: A path coming from the diverging side of a switch may only leave on the other side, and vice versa. A path entering a crossing from one of its four neighbors can only exit at its diagonal counterpart neighbor. A path can only turn around at buffer stop and station vertices. To decide on which neighbors we expose to the Dijkstra algorithm, additional information about where the current vertex was entered from is required. For example, if the node currently considered by the algorithm is a switch, we need to be aware whether the path which is currently considered comes in at the diverging side or the non-diverging side. If the last vertex of the path is on the diverging side of the switch, we want to only expose the one non-diverging side, and vice versa. This way, we can prevent the algorithm from returning illegal paths. Whether a neighbor is exposed determines whether Dijkstra can recognize the vertex during the current iteration. I.e. the algorithm can not add non-exposed vertices to the heap or update their information. The biggest hurdle to this approach to railroad routing is obtaining the necessary information to make a decision on whether to expose a neighbor. More precisely, we want to know which neighbor a vertex was entered from. Our solution to this hurdle is logically duplicating the graph to four copies, which is rather memory and runtime intensive solution. Why such a solution is necessary, is explained in the next paragraph.

One may assume that the Dijkstra's predecessor array is enough, however, a simple example can illustrate this is not the case. Figure 6.1 shows a railroad layout where using the predecessor does not result in the shortest path. It shows a layout similar to the number "eight" where two different paths are possible to reach target vertex ⑦ from source ①. The arcs between edges indicate an acute angle, where traveling along is not permitted. In the middle of the illustration lies vertex ④, which is a crossing and therefore only allows diagonal travel through it. So, depending on whether we enter ④ from vertex ② or ③ we get a different path. Since Dijkstra's algorithm is greedy, the algorithm will get to ④ over vertex ③ first, which brings in a path length of 2, compared to length 6, which the upper path would have. Afterwards when the algorithm starts discovering the neighbors of ④, since its predecessor is now ③, it is only possible it only possible to continue over the longer path via ⑤. The vertex with lowest distance on the stack is then ②, which tries to add ④ to the heap but can not do so because ④ is already visited. Thus, the algorithm will only be able to find the inferior path (1,3,4,5,7) with length 12 compared to (1,2,4,6,7) with length 8. If one opts

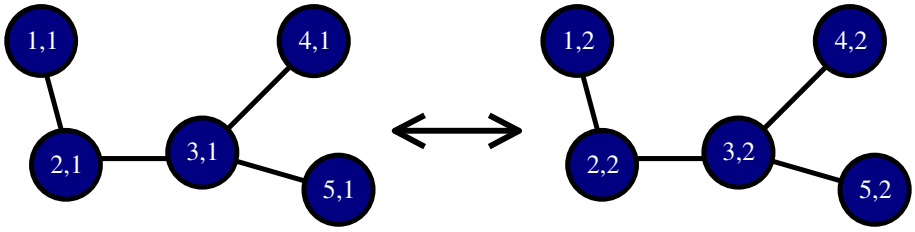


Figure 6.2: Graph duplication as a mean to give Dijkstra's algorithm more state. This information is then used to prevent illegal paths.

for removing the visited set, it is trivial to construct a simple cyclic example where the algorithm does not terminate. Cycles are also the reason one can not count the number of times a vertex was visited, as using a cycle the vertex can be visited multiple times from the same neighbor.

Our solution, which is inspired by the approach of Herrmann [Her06] modeling the allowed paths using a dual graph, is demonstrated in Figure 6.2. The figure shows a logical duplication of a graph to add state to the algorithm. When using logically duplicated graphs, we no longer consider vertices in the base graph, but instead their counterparts in the logical graphs. In these logical graphs, vertices are described by tuples. The first number of the tuple is the vertex ID and the second number is ID of the graph the vertex belongs to. For example, if the current vertex of our Dijkstra iteration is (34,2) this means the algorithm is currently handling the node with ID 34 on the graph with ID 2. To the Dijkstra algorithm (34,1) and (34,2) appear as completely different vertices, as they are on different logical graphs. The graph ID can be used to add arbitrary information to a vertex, but in our case it indicates which neighbor the node was visited from. What graph ID the routing algorithm uses is determined by which neighbor it enters the vertex from, e.g. if it visits a vertex from neighbor 2 it considers the vertex at graph ID 2. The number of logical graphs needed, corresponds to the bits of information required. In the case of our routing algorithm, the crossing vertex is the type with the highest number of neighbors, requiring two bits to differentiate between its four neighbors. Four logical graphs are sufficient to encode these two bits of information, so this is what the simple and multi-dimensional routing algorithm use. Implementing logical graphs just as they are explained here is not practical though. In reality, we only need to keep the base graph in memory, as logical graphs are a logical construct. Since the only information that is altered for the logical graphs are vertex IDs, it is simply not necessary to use additional memory to copy the whole graph. Instead, when adding vertices to the Dijkstra heap during the discovery step, we add a vertex from the logical graph containing the information which neighbor it was discovered from. Then, whenever the Dijkstra starts its iteration on a new vertex, we can read the second element of the tuple and determine which neighbors are to be exposed.

The last thing to discuss about the simple routing algorithm is its method of optimizing for path distinctness and infrequent changes of track. Both are achieved using penalties, i.e. increases to the edge weight, which make these edges less desirable for Dijkstra to use. The duplicate edge penalty is applied statically. For the first alternative, no penalty is applied, all edge weights are set to the duration it takes a train to travel along it at the maximum allowed speed. Using time as a measure for weight, instead of the more commonly used length, allows us to optimize paths for time which is more sensible for railroad planning. All of the edges this calculated path uses are added to a used edge set, so that the algorithm can apply a penalty to them. When starting the calculation of the second alternative this penalty has to be applied. This is done by copying the whole graph, and

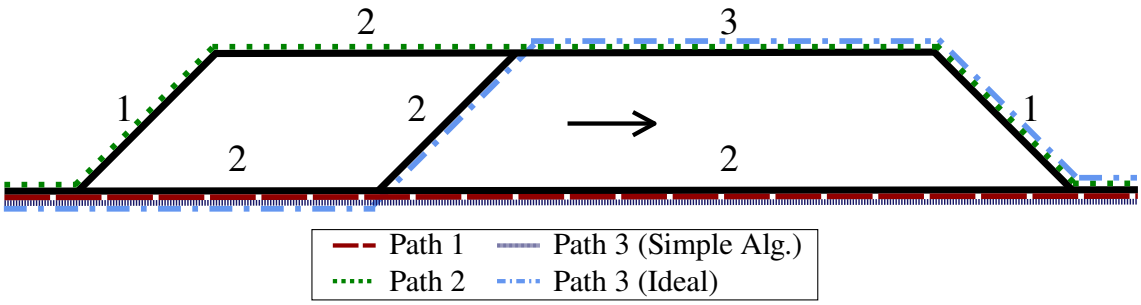


Figure 6.3: Shortcomings of the alternatives calculated by the simple algorithm. Path 3 shows that in situations with a lot of used edges, the simple algorithm delivers sub-ideal results.

changing the weight of all edges to apply the potential penalty. The desired amount of penalty is given as an input parameter, which represents a factor that is multiplied with the weight of the edge once it is contained in the used set. The algorithm then calculates the second alternative, on the graph with the applied penalty. After finishing the calculation all used edges are added to the used set and the previous steps are repeated for following alternatives. Unlike the duplicated edge penalty, the penalty for changes of track is applied dynamically. This means that, the penalty is not actually applied to the whole graph, but added later during routing. The reason why this penalty has to be dynamic, is that the weight of an edge changes depending on context. If a path enters a switch and leaves at diverging path which is not straightest, a penalty should be applied. However when reversing this scenario, i.e. enter the switch from the diverging path, it should not be applied. The penalty is dynamically added as a constant. The input parameter specifies how many seconds of penalty should be applied in case of a track switch.

6.2 Multi-Dimensional routing algorithm

The multi-dimensional routing algorithm was proposed once we evaluated the simple routing algorithm and found that it did not always produce ideal results. Figure 6.3 illustrates a case where this holds true. In the given example three different alternatives are supposed to be calculated for a track section. The penalty factor for duplicated edges is captured by an arbitrary value d. The first, and therefore shortest, path which is calculated by the algorithm will use both lower edges with a total length of 3. Path number two depends on the value of d. If $d > 2$, which is the more reasonable option, path 2 diverges at the first switch and travels all the way over the upper edges. With lower values for d path 1 will be the only alternative calculated. The issue with the algorithm is apparent when calculating path 3. By this point, all edges except for the second switch have already been visited and added to the $used$ set. For the best diversity in our result, one would add a way using this second switch, the only possible path which has not been added to the set yet. However this ideal path has a length of $2d + 2 + 3d + d = 6d + 2$, while traveling on the quickest path again now has length $2d + 2d = 4d$. No matter the d, $6d + 2 > 4d$ always holds for positive values of d, thus there is no possible parameter to get an ideal result set using the simple algorithm. The following conclusion can be drawn from this issue: When checking how similar a path is, we have to compare it to each existing path individually, instead of generalizing all existing paths into one set. Using this conclusion we propose a second algorithm which mitigates this issue at the cost of higher complexity as well as run time and memory demands.

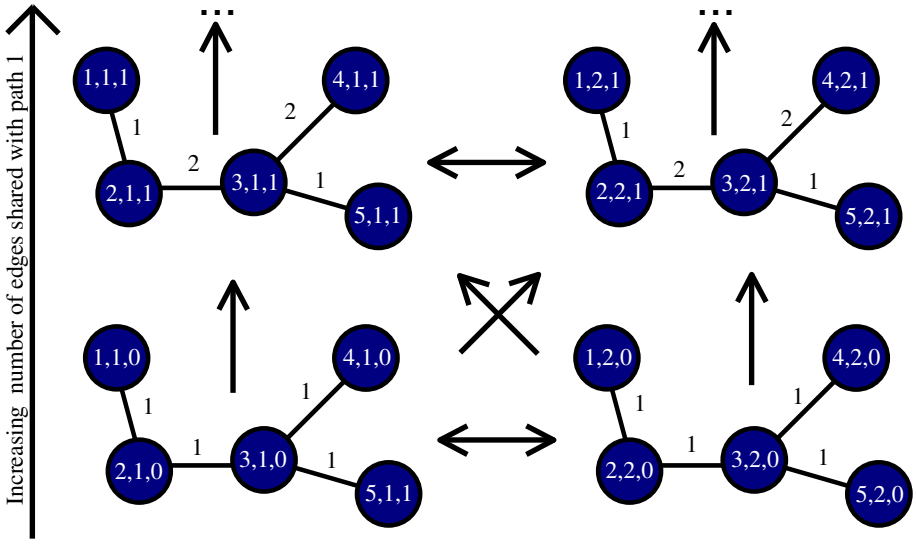


Figure 6.4: Using a one-dimensional graph to model edge duplication. There are infinitely many graphs when following the dimension upwards.

The multi-dimensional algorithm is an alteration of the simple routing algorithm, that utilizes a different method for applying the duplicated edge penalty. The goal is to maintain information on how much distance is shared with each specific alternative to apply more punishment if a lot of edges are shared with a single one. As for how to store the information, our solution builds on the earlier introduced concept of logical graphs. It has already been established that logical graphs can be used to store additional information, although for a limited number of bits. Since, we do not know an upper bound to the number of shared edges we require this number to be infinite. Thus, a third number was added to the vertex tuple, which indicates the number of shared edges shared with the first alternative. Figure 6.4 illustrates such a one-dimensional graph. Increasing the third number allows access infinitely many logical graphs, with each of them having their own edge weights, so penalties can be applied based on the position in the dimension. If we want to calculate the third alternative we need to increase the graph dimension to two, so vertices are now tuples with four elements. The first element indicates the vertex ID in the base graph, the second element the predecessor neighbor, the third element the number of edges shared with alternative one and the fourth element edges shared with alternative two. In general, calculating the n-th alternative requires us to use a $n-1$ dimensional graph for routing. The dimensional position is used to apply punishment more dynamically. If an edge is shared with alternative n the algorithm reads the dimensional position, which measure shared edges with that alternative, and adds a penalty of $pos(n) * W(e)$.

Since every single dimension in our dimensional graph model extends infinitely, there also is an infinite number of vertices that can possibly be visited by a routing algorithm. This causes a loss of the termination guarantee Dijkstra's routing algorithm has on non-dimensional graphs. Termination is still guaranteed, if a path between source and target exists. This is because all edges have a positive weight, thus the strictly monotonically increasing weight of the current node will eventually reach the distance of the target and conclude the algorithm. But, if there is no path between source and target and therefore no routing solution, termination is not guaranteed. Since there is an infinite amount of nodes to explore the algorithm follows cycles infinitely. This termination problem can be

fixed and algorithm performance be improved by adding a check to stop expanding a dominated path.

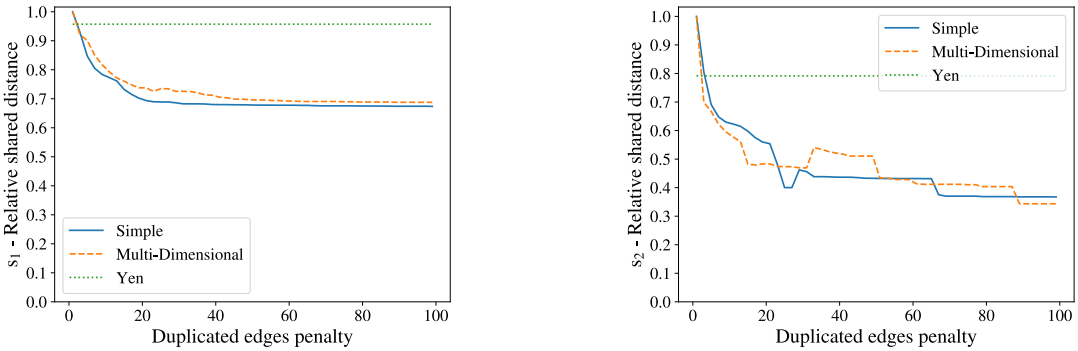
$$\begin{aligned} & dominated((v_i d, dir, d_1, d_2, ...)) \\ & \Leftrightarrow \exists (id', dir', d_1', d_2', ...) \in distance : \\ & id = id' \wedge dir = dir' \wedge distance(v) < distance(v') \wedge d_1' \leq d_1 \wedge d_2' \leq d_2 \wedge ... \end{aligned} \tag{6.1}$$

If *dominated* evaluates to true, we can safely conclude that there exists another path that superior the current one, so we can decide to not do any follow up calculations for it. The equation looks for the existence of a path to the same base vertex in another logical graph, that is guaranteed to be part of a superior path to the target. It does so by checking the *distance* array, which contains all vertices and their corresponding distance. The equation checks for the comparability of the vertices, by checking if their vertex ID is identical, as well as their graph ID, i.e. if both ways enter it in the same direction. If this holds true, we know that we found a path to a vertex which offers exactly the same routing opportunities. The following conditions can then check for the superiority of the graph. To be dominating the vertex needs to be superior in every value. The distance of the vertex needs to be lower and the graph needs to be equal or at a lower position in every single dimension, i.e. will receive less penalties in the future. Checking this condition on every iteration allows us to restore Dijkstra's guarantee of termination, as well improve runtime, as the algorithm no longer follows cycles.

7 Evaluation

In order to verify and quantify the improvements experienced using the simple and multi-dimensional routing algorithm, we performed an empirical evaluation. Goal of the assessment is to investigate the capability of the algorithms to maintain the quality aspects presented in our problem statement (Section 2.1). In summary, we evaluate for the distinctness of the alternative routes, the quantity of track changes, as well as the length and travel time of the routes. The evaluation data set was established by randomly selecting active German train lines. 15 lines were picked in total, subdivided into 10 regional lines (lines labeled RB, RE, IRE, MEX) and 5 long-distance lines (IC, ICE). The lines were broken up into these two categories, because regional lines are generally shorter than 100 kilometers, while long-distance lines have multiple hundreds of kilometers, so both can be seen as independent use-cases. For everything but the runtime evaluation, the shown statistics are an average across all lines of the category. Using our Python reference implementation, we calculated a total of three alternative routes for each line. The maximum vehicle speed for the regional evaluation was set to 160 km/h and 250 km/h for the long-distance lines. The evaluation was performed on a dual socket AMD EPYC 7451 system, which was restricted to use two of eight available NUMA nodes. This restriction does not affect performance as two nodes allow the application to use 12 physical cores, which satisfies our maximum degree of parallelism, which is 10. To run the assessment, we had to manually modify the algorithms slightly. To be specific, the feature that detects an alternative having a path which is unchanged from one of its predecessors stopping further calculations was removed, as to not falsify the results.

Along with both of our algorithms, we evaluated Yen's k-shortest path algorithm. Yen serves as an example for an existing common set routing algorithm, but plays an especially important role as it is the routing algorithm Falk et al. [FGD+21] use. Thus, the comparisons to it allow us to estimate the potential benefit their routing and scheduling solution can achieve using our candidate set. The algorithm is routing independent, so it can be run on top of an arbitrary shortest path algorithm. This allows us to use it with our existing Dijkstra implementation, used in both of our algorithms, which was modified not to take any illegal paths. For implementation, we used an existing Python implementation of the algorithm provided by network analysis library NetworkX [dev]. However, this implementation had to be extended, as Yen's algorithm does not allow for specifying intermediate stops. We solved this in a similar fashion to our routing algorithms, by splitting lines up into multiple single-source-single-target shortest path problems, which Yen's algorithm can solve individually. The alternative paths are then determined by concatenating the paths calculated in each sub-problem to receive an overall result. For the first alternative, we concatenate the first path calculated in each sub-problem, for the second alternative the second ones, etc. Note, that this means the second alternative returned by our implementation of Yen, is not actually the second shortest path, making it strictly speaking not a k-shortest path algorithm. However, considering how similar the different paths determined by Yen's algorithm are, this is a welcome trait as it allows more sensible comparison between it and our algorithms.

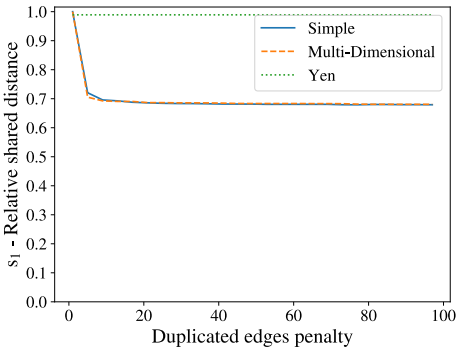


(a) s_1 - Relative shared distance

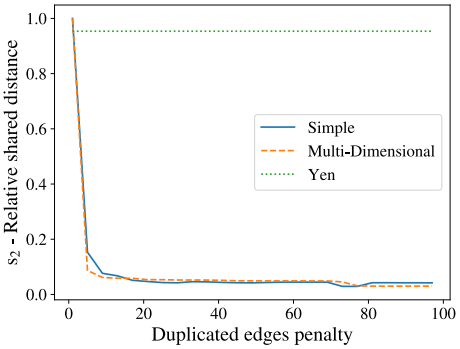
(b) s_2 - Relative shared distance

Figure 7.1: Relative shared distance between alternatives (regional traffic). The less shared distance, the more diverse the resulting alternatives are. s_1 compares against all other paths as a unity, s_2 compares the similarity on a per path basis.

Figure 7.1 indicates how distinct the alternatives calculated for the lines are. High distinctness is an important quality of the candidate path set, as it grants a train routing problem algorithm more options to avoid conflicts. The distinctness of the paths is determined by scoring the distance shared between them, so lower is better in these charts. Figure 7.1a determines the shared distance of the alternatives using metric s_1, which was presented in Section 2.1. In short, s_1 measures the relative shared distance by determining whether each edge is also present in any other alternative. The chart plots the input duplicate edge penalty against the s_1 value of the result set. Both the simple and multi-dimensional algorithm start at a shared distance of 100% for parameter 1. At that value, no penalty is applied during the routing process, thus both algorithms return the shortest path thrice. For penalties bigger than 1, both algorithms instantly overtake Yen's algorithm, which has a high degree of alternative similarity with 96% shared distance. This does not come as much of a surprise, since the algorithm was never designed with alternative distinctness in mind, rather the opposite is the case. As for our algorithms, there is no outstanding difference between the two, but the simple routing algorithm consistently outperforms the multi-dimensional algorithm. While perhaps seeming odd at first, considering the multi-dimensional algorithm is more sophisticated, the superiority of the simple algorithm can be explained. Our metric s_1 is identical to the optimization target of the simple algorithm, while the multi-dimensional algorithm optimizes towards a different goal, one that is more akin to s_2. This also explains why the curve of the simple algorithm is monotonously falling, while multi-dimensional has small bumps. Metric s_2, which measures distinctness in a different way, is plotted in Figure 7.1b. The main difference compared to s_1 is that s_2 checks for similarity on a per path basis and punishes the score if two particular paths share a lot of edges. The results of this metric seem shifted downwards by about 20% compared to s_2, which is caused by the fact that s_2 is inherently lesser than s_1. s_2 determines the maximum shared distance between a path and any single other one, so any duplications here will also be duplications according to s_1. Overall, both curves are a lot more bumpy compared to the previous plot. Unlike s_1, which only averages, s_2 additionally uses the maximum function, which is vulnerable to outliers in the data causing sudden jumps. The high amplitude of the bumps further reflects this trait, small changes in routing can have a big impact on the way s_2 scores the result. Where the bumps occur can be attributed to chance. The bumps can also go up as s_2 is not a direct optimization goal to



(a) s_1 - Relative shared distance

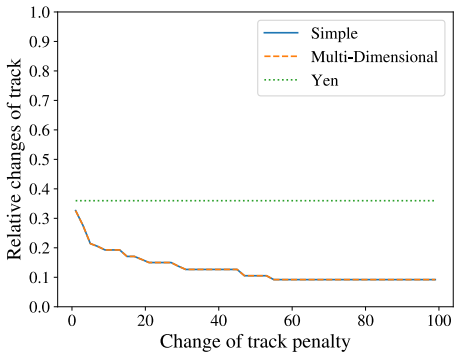


(b) s_2 - Relative shared distance

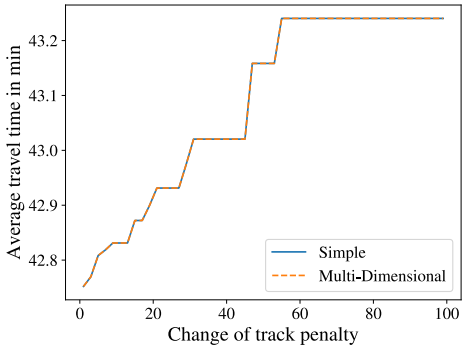
Figure 7.2: Relative shared distance between alternatives (long-distance traffic). The performance of our algorithm assimilates and steady state is reached more quickly.

either algorithm, so the alignment of the actual goal and s_2 changes. s_2 is neither the optimization goal of the simple algorithm nor the multi-dimensional algorithm, so some solutions end up aligning with s_2 well and some do not. For low values of the duplication penalty, the multi-dimensional algorithm outperforms the simple algorithm, otherwise which one is better is a toss up. What is noteworthy though, is that we can still observe improvements for higher penalties like 65 for the simple algorithm and 90 for the multi-dimensional algorithm, while progress in s_2 stagnates much more quickly.

Figure 7.2 shows plots of the same metrics, but this time for our 5 long-distance lines. The most important thing we can take away from Figure 7.2a plotting s_1, is that longer routes align the performance of both of our algorithms. This is made clear by the fact that the performance of Yen did not converge towards the same result, but away from it raising all the way to 99%. To understand why this is the case, one has to consider how the algorithm works. In short, Yen's algorithm calculates alternatives by removing some edges from the graph, which are part of previous alternatives. The first alternative is the shortest path, the second alternative is a deviation where any one edge is made inaccessible, a second edges is removed for the third alternative, and so on. Since the number of alternatives for the evaluation is fixed, we know that the alternatives only delete one or two edges, respectively. Therefore, the absolute distinctness for these alternatives is constant and plays a smaller role, the longer the planned line is. Figure 7.2b confirms two observations we have made for the regional traffic lines. The multi-dimensional algorithm still outperforms the simple algorithm for low penalties, although the gap is shrinking much sooner now. Otherwise, both algorithms still deliver comparable performance. What changed however, in both Figure 7.2a and Figure 7.2b, is how quickly the algorithms reach a steady result for the long-distance lines. Another clear difference compared to the regional evaluation is how low s_2 is in comparison to s_1. Since s_2 and s_1 have intersections in how they function, we can draw a conclusion from that. In long-distance lines, routes still use similar tracks a lot of the time, but individual combinations in which they use them are much more unique. This is down to the fact that longer distances between intermediate stops also mean more possible combinations of the tracks in between.



(a) Relative number of track changes. Simple and multi-dimensional algorithm get equal results because they apply the same penalty.

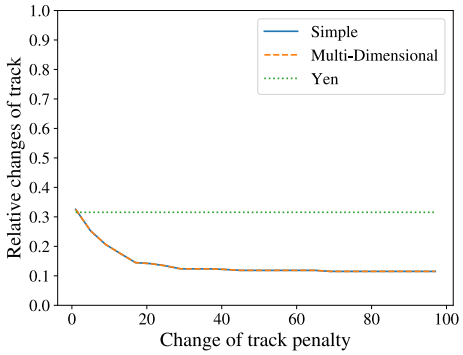


(b) Average route travel time with respect to change of track penalty. The chart behaves anti-proportional because of conflicting optimization goals.

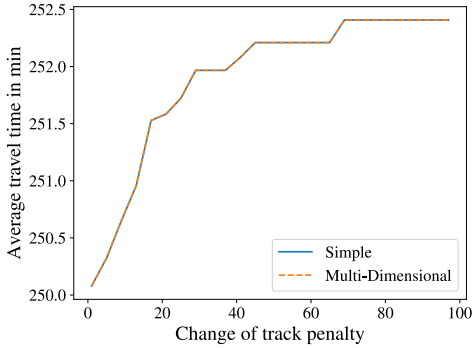
Figure 7.3: Evaluation of track change penalty and its impact on travel time (regional traffic).

Figure 7.3a plots another quality aspect, the relative number of track changes. The y-axis measures the ratio of switches, where the path does not follow a straight way through the switch, but leaves the switch at a sharper angle. Generally, a lower value is favorable here because frequent track changes slow down a train and increase chances of conflicts with oncoming trains. However, track changes are part of the design of train routes, especially in Europe [LLER11], so this value never reaches zero. For the measurement, only switches where we enter on the non-divergent side are considered. If we counted all switches, in 66% of cases the switch would be entered on the divergent side, only allowing the train to exit on the non-diverging side, thus deflating the metric. Noteworthy is, that the curves of the simple and multi-dimensional algorithm are identical in this chart. Since this statistic is supposed to isolate the changes caused by the change of track penalty, the penalty for duplicate edges was set to zero. As the simple and multi-dimensional algorithms only differ in their handling of duplicate edges, they produce identical results in such a scenario. Without any penalty, Yen's algorithm performs slightly worse than our algorithms in this statistic. The difference can be seen as up to chance, since Yen optimizes for path shortness and our algorithms optimize for low travel duration. Increasing the penalty, we observe steep changes up to a penalty of 5 seconds, followed by linear improvements up to penalty 55 and steady state henceforth. Since we logically established that the optimization goals, low travel time, few track changes and low path similarity are in conflict with each other, it makes sense to evaluate impact on those other two metrics. Figure 7.3b evaluates the impact of the penalty on travel time. Since these two compete, the curve behaves in an anti-proportional way to the last chart, each bump upward corresponds to a bump downward in the last chart. What is relevant in this chart is the amplitude of the bumps on both charts. Ideally, we want to see large decreases in the left chart with small increases in the left chart. Overall, we see a maximum increase in travel time by 1.1% which is negligible compared to the impact the duplicate edge penalty has.

When calculating the long-distance routes (Figure 7.4a) the track change statistic overall looks similar to the one in regional traffic. Without any punishment, the algorithms still perform mostly similar to Yen's algorithm, although they are a bit closer now and Yen outperforms our algorithms. The change is up to chance, since Yen calculates the shortest route and our algorithms calculate the



(a) Relative number of track changes. The curve reaches steady results faster.



(b) Average route travel time with respect to change of track penalty. The impact on travel time is low, so we can conclude there is little competition between the two.

Figure 7.4: Evaluation of track change penalty and its impact on travel time (long-distance traffic). Simple and multi-dimensional algorithm get equal results because they apply the same penalty.

quickest route. Just as observed on the duplicated edge penalty, the track change penalty achieves a steady value more quickly compared to the regional routes. We attribute this observation to the fact that long-distance lines offer a greater number of possibilities for avoiding duplicated edges / track changes. Their gap between intermediate stops is comparatively large, thus giving the algorithms more freedom to chose routes between these stops. Also, the intermediate stops of a long-distance line are larger stations with many more tracks and platforms. Therefore, the algorithms are offered a much greater set of possibilities to avoid penalties more easily. Figure 7.4b plots the impact of the penalty on the average travel time of the calculated routes, this time for long-distance traffic. The impact of the penalty is even smaller here, with an overall increase of 0.9% within the shown interval. Both the regional and the long-distance statistic lead to the conclusion that the competition between path travel time and changes of track is low. The logical conclusion would be that the track change optimization goal mainly competes with the goal of low shared distance.

Figure 7.5 confirms this hypothesis. The figure plots the change in relative track changes as we increase the duplicate edge penalty. As it is based on the regional traffic dataset, this plot is best compared with Figure 7.3a. For this statistic, we fixed the change of track penalty to 80, which is where both regional and long-distance had reached a steady value. On the x-axis, we increase the duplicated edges penalty, in an interval of [1,20]. Even for low values of the duplicated edge penalty one can observe a drastic increase in changes of track. This leads to the conclusion that the duplicated edge penalty is a much more sensitive penalty than changes of track. Even small changes can have big impacts, so precise tuning is required.

Figure 7.6 increases both penalties to investigate changes to route length and travel time in long-distance traffic. Both metrics are a high priority for train timetabling, as customers do not want to travel for any longer than necessary. A long route length generally also increases the chance of conflict with routes of another line. This in turn, forces the algorithm to produce more diverse results in order to avoid the conflict, a snowball effect which should be avoided. The first apparent observation is that both graphics look almost identical. The similarity can be attributed to the layout

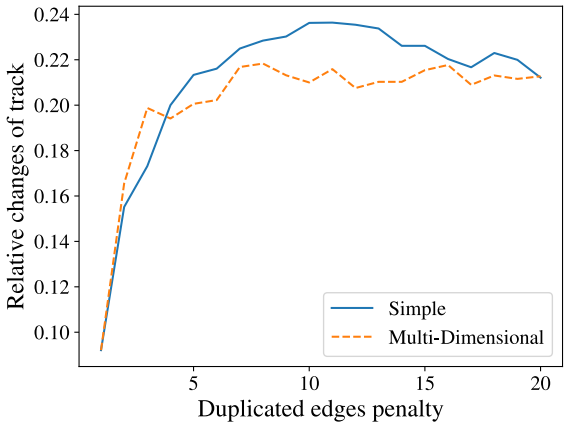
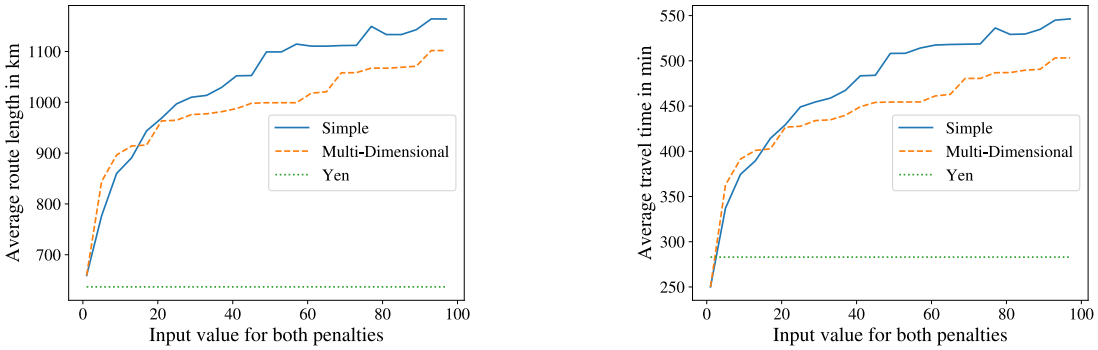


Figure 7.5: Impact of duplicate edge penalty on changes of track (regional traffic). The large impact shows that careful balancing needs to be done between both penalties. [change of track penalty = 80]

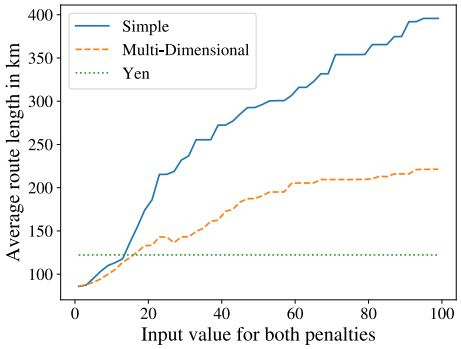


(a) Average route length

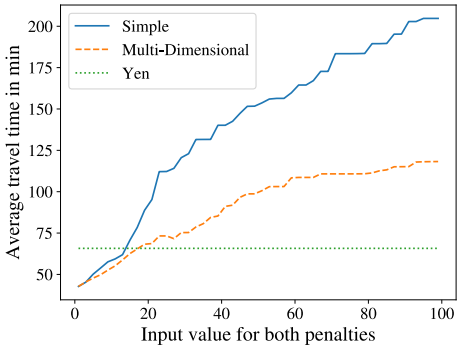
(b) Average travel time

Figure 7.6: Average route length and travel time (long-distance traffic).

of train tracks. High-speed lines tend to follow an especially straight line, which is part of why they allow traveling at high speeds in the first place. Thus, in many cases, the shortest path is also the quickest path, leading to identical results in those metrics. Figure 7.6a displays Yen outperforming the simple and multi-dimensional algorithm in terms of route shortness by at least 3%, even without any penalties applied This is again, because Yen's algorithm optimizes for route shortness, unlike the algorithms proposed in this paper that optimize for travel time. For low penalty values up until 15, the multi-dimensional algorithm delivers longer routes than the simple algorithm. This is part of why, in long-distance tests, the multi-dimensional algorithm achieves better shared distance results for low penalties, compared to the simple algorithm. We previously concluded multi-dimensional and simple routing assimilate for longer lines. The figure confirms this assumption. The algorithms differ in how they scale with the penalties, but overall the results of the multi-dimensional algorithm can be achieved with the simple algorithm by adjusting the penalties, and vice versa. In conclusion, for this type of route, there is no inherent difference between the capabilities of both algorithms.



(a) Average route length

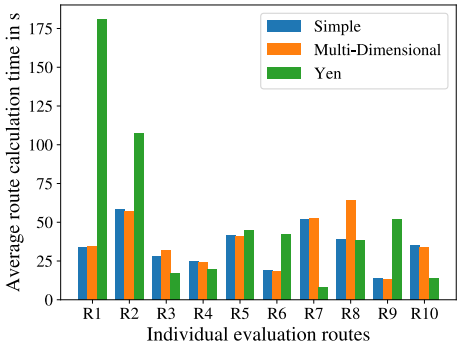


(b) Average route travel time

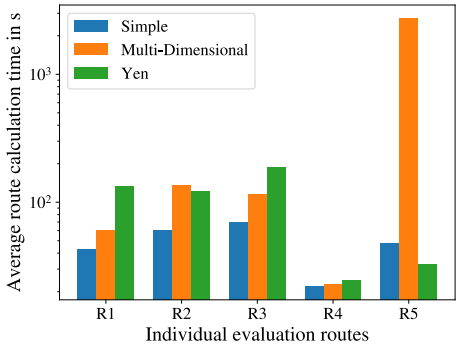
Figure 7.7: Average route length and travel time (regional traffic). Simple and multi-dimensional algorithm get equal results because they apply the same penalty.

The same can not be said for regional lines, which are plotted in Figure 7.7. Here, the difference between both algorithms is more noticeable. For regional lines, the multi-dimensional algorithm produces results which are consistently shorter and quicker. Up until penalty 17, the simple algorithm produces noticeably longer routes for its result set. For higher penalties, this gap starts widening the higher the penalty goes, with the multi-dimensional algorithm ending up with about half the distance of the simple algorithm. This, along with comparable shared distance performance, makes it the clear winner in terms of result quality for high penalties. For low penalties, which algorithm is more suitable depends on the use-case. Algorithms benefiting from a low s_1 achieve better results using the simple algorithm, those benefiting from s_2 likely prefer the superior performance of the multi-dimensional algorithm. Interestingly, Yen's algorithm does not produce the shortest paths anymore, even going as far as having over 50% longer results. The cause of this phenomenon are outliers in the data. Compared to the long-distance lines, which leave the algorithm with a lot of possible routes, routing on the smaller scale regional lines proved to be a lot harder. Such track constructs cause an increase in path length up to tenfold and therefore drag up the average by a lot. The outliers occur where the track between two intermediate stops is hose-like, i.e. does not allow for any track changes. The biggest offender occurs on route 1, where two stations are linked by two railroad tracks, which are both uni-directional, thus only one of them is usable. Yen's algorithm is unable to produce a sensible alternative here and ends up generating a path with a distance that is tenfold the shortest one.

These outliers also show up in our runtime performance evaluation (Figure 7.8). Time measurement for each of the calculations was averaged over $N = 10$ sample runs, with a relative standard deviation of $\sigma < 0.5\%$. The plots show each of the different test routes on the x-axis, the y-axis plots the time needed to calculate the set of three paths, so lower results are considered better. All tests were performed with 15 as the penalty parameter for both changes of track, and shared edges. Which routes contain outlier situations for Yen's algorithm is pretty clear from this chart, regional lines R1, R2 and R9. Both contain a section where the track between two intermediate stops is hose-like, i.e. does not allow for any track changes. For example, two intermediate stations are linked by two railroad tracks, which are both uni-directional, thus only one of them is usable. The problem is that for the second and following alternatives, Yen's algorithm always deletes at least one edge, so it is always forced to find a distinct path, even when there is no sensible one to be found. The tenfold



(a) Runtime per route in regional traffic



(b) Runtime per route in long-distance traffic. Logarithmic scale.

Figure 7.8: Runtime per route. [$N = 10, \sigma < 0.5\%$, penalties = 15]

increase in route distance goes hand in hand with a large increase in runtime. But, there are routes, where Yen's run time trumps all other algorithms, especially R7 and R10 of the regional evaluation. This is because Yen does not avoid shared distance the same way our algorithms do. For these routes, Yen calculates a set of 3 very short paths, whilst the simple and multi-dimensional algorithm take longer paths to produce unique alternatives. For the regional traffic lines, we can reach the conclusion, that the main factor influencing runtime is not inherent algorithm efficiency, but how long the produced results paths are. The same conclusion can not be reached for long-distance routes as plotted in Figure 7.8b on a logarithmic scale. Here, the simple routing algorithm ends up as a pretty clear winner, having the lowest runtim in all routes except for R5. A huge outlier is the multi-dimensional algorithm for route 5. The bad runtime can be attributed to scaling. The multi-dimensional algorithm does not scale well for routes with long distances between intermediate stops, which is the case in R5.

Overall, we conclude that both the simple routing algorithm and the multi-dimensional algorithm are capable of producing results which are significantly improved compared to Yen's algorithm. This holds true for all quality aspects, shared distance, track changes, travel time, and even route length in some scenarios. Which type of algorithm is recommendable to use, depends on the calculated route and use-case. If the calculated route is lengthy, especially having parts with a long distance between intermediate stations, the simple routing algorithm should be considered. The difference in result quality for both algorithms in this use-case is negligible, but the simple algorithm displays superior runtime. For short routes the recommendation depends on the use case. Most algorithms should benefit from the superior s_2 shared distance, so the multi-dimensional algorithm should be considered. However, some algorithms may not need paths according to the stricter definition of uniqueness by s_2, and profit from the slightly superior s_1 performance of the simple algorithm. In any case it is important to note, that in order to achieve ideal results, precise tuning of the parameters is required, because all the different optimization goals are conflicting. This conflict requires us to balance the parameters against each other carefully. How achievable this balance actually is, when running the algorithm on numerous diverse routes to process a large region, remains to be seen. It is conceivable that extensions could be developed in the future, which ease balancing of the parameters using basic line, graph and route metrics.

8 Conclusion and Outlook

To conclude the thesis, we want to recall important topics within it. First, we introduced the reader to the train routing problem, in which routes are assigned to train lines, and the train scheduling problem in which schedules are assigned to train lines. We pointed out that the computer network routing and scheduling solution of Falk et al. [FGD+21] may solve these problems efficiently, but requires a sensible set of candidate paths. The thesis then explained that the railroad system is organized into safety blocks, because of its tolerance for failures and technological ease of implementation. The implementation of the blocks is done via various types of signals which inform the driver about the clearance status of upcoming blocks. In the next step, we explained how we transform the geographic OpenStreetMap data set into a graph which only contains the necessary railroad information. We also showed that during this procedure, we can add additional information about the topology of the network to improve routing, e.g. the divergent and non-divergent side of a switch. Next, we propose two routing algorithms, the simple and multi-dimensional routing algorithm. Both algorithms have the goal to maximize the uniqueness of a path, but different definitions of uniqueness allow for different algorithms. The simple algorithm tries to avoid all edges which are also contained by any another path. It does not distinguish how often the edge was used, or which path uses it. The multi-dimensional algorithm is an alternative way to find unique paths, which determines uniqueness on a per path basis. We showed how we can use the information which was tagged on to the graph to improve routing results. These improvements range from essential, like preventing illegal paths, to supplementary, like the default way through a switch. Finally, we evaluated both algorithms against each other and an implementation of Yen's k-shortest path algorithm. The evaluations showed that the difference between the simple and the multi-dimensional algorithm is not big. Both algorithms are capable of improving on Yen regarding shared distance, changes of track and result set shortness substantially. But, we also observed that, since all of these goals are competing with each other, the parameters need to be balanced carefully to achieve optimal results.

Outlook

This thesis does not conclude the potential research on the simple and multi-dimensional algorithm. Most important is that this thesis was conducted as part of an effort to adopt the computer network scheduling algorithms of Falk et al. [FGD+21] into the railroad domain. However, integration between their algorithms and herein proposed candidate path routing algorithms has yet to be conducted. Only after this has been done can we evaluate the actual benefits that can be seen when using this algorithm in a routing / scheduling process. In order to run this algorithm on huge data sets, runtime performance optimizations of the implementation specifically should be considered. Since the main purpose of our Python implementation is to serve as a proof concept, little attention to performance optimization was paid. As the algorithm is working on large sets of data, a compiled

language like C should be considered. One could study the effects candidate routing solutions akin to our could have to improve computer network scheduling, effectively reversing the translation in domain.

Then, there are also improvements which can be done to algorithms itself. When routing on bidirectional tracks, we make the assumption that it is sufficient to consider only the signals in one direction and view them as if they are able to signal for both directions. This eliminates the need for signals in the opposite direction, which is why they are not considered. However, in reality, signals in a bidirectional track are often physically separate, which means they can be shifted in position from the opposite signals. In a theoretical model, this is a sane assumption, because the distance between two signals is the same and precise position is not important. But, when using the algorithm in the real world, we would want to have our graph precisely model the real signal constellation, i.e. map real signals to signals in the graph one to one.

A possible extension for the multi-dimensional algorithm specifically are continuous dimensions. The dimensions we use for the multi-dimensional algorithm are discrete, i.e. only allow increases by integer numbers. The dimensions count the number of shared edges, but since edges can have varying length, this is not the most accurate way of measuring how similar paths are. If one were to adapt the algorithm to use continuous dimensions, the shared distance could be used as a dimension and the results obtained may become more accurate.

Bibliography

[Bun21] Bundesnetzagentur für Elektrizität, Gas, Telekommunikation, Post und Eisenbahnen. *Marktuntersuchung Eisenbahnen 2020*. 2021 (cit. on p. 23).

[Bur05] D. M. Burkolter. "Capacity of railways in station areas using Petri Nets". Artwork Size: 147 p. Medium: application/pdf Pages: 147 p. PhD thesis. ETH Zurich, 2005. DOI: 10.3929/ETHZ-A-005060957. URL: http://hdl.handle.net/20.500.11850/46806 (visited on 12/03/2021) (cit. on p. 16).

[CBH+09] G. Caimi, D. Burkolter, T. Herrmann, F. Chudak, M. Laumanns. "Design of a Railway Scheduling Model for Dense Services". In: *Networks and Spatial Economics* 9 (Feb. 2009), pp. 25–46. DOI: 10.1007/s11067-008-9091-6 (cit. on p. 17).

[CBH05] G. Caimi, D. Burkolter, T. Herrmann. "Finding Delay-Tolerant Train Routings through Stations". In: *Operations Research Proceedings 2004*. Ed. by H. Fleuren, D. den Hertog, P. Kort. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 136–143. ISBN: 978-3-540-27679-1 (cit. on p. 16).

[CD07] S. Cornelsen, G. Di Stefano. "Track assignment". In: *Journal of Discrete Algorithms*. 2004 Symposium on String Processing and Information Retrieval 5.2 (June 1, 2007), pp. 250–261. ISSN: 1570-8667. DOI: 10.1016/j.jda.2006.05.001. URL: https://www.sciencedirect.com/science/article/pii/S1570866706000475 (visited on 12/03/2021) (cit. on p. 16).

[CGT07] A. Caprara, L. Galli, P. Toth. "04. Solution of the Train Platforming Problem". In: *Transportation Science* 45 (Jan. 2007). DOI: 10.2307/23017703 (cit. on p. 16).

[DB] DB Netz AG. *Trassenfinder*. URL: https://www.trassenfinder.de (cit. on p. 23).

[DB 19] DB Netz AG Zentrale Betriebsverfahren. *Fahrdienstvorschrift, Richtlinie 408*. 2019 (cit. on p. 23).

[Deu] Deutsche Bahn AG. *Jobs bei der DB, Fahrdienstleiter (w/m/d)*. URL: https://karriere.deutschebahn.com/karriere-de/jobs/schueler/ausbildung/fahrdienstleiter-deine-ausbildung-2650412 (cit. on p. 23).

[Deu20] Deutscher Bahnkunden-Verband e. V. *Eisenbahninfrastruktur Gesamtübersicht Deutsche Regionaleisenbahn Gruppe*. 2020 (cit. on p. 23).

[Deu21] Deutsche Bahn AG, Investor Relations und Sustainable Finance, Berlin. *DB Netz AG Geschäftsbericht 2020*. 2021 (cit. on p. 23).

[dev] N. developers. *NetworkX*. URL: https://networkx.org/ (visited on 12/03/2021) (cit. on p. 41).

[Edm65] J. Edmonds. "Minimum partition of a matroid into independent subsets". In: *J. Res. Nat. Bur. Standards Sect. B* 69 (1965), pp. 67–72 (cit. on p. 15).

[Eis] Eisenbahn-Bundesamt. *Eisenbahnunternehmen*. URL: https://www.eba.bund.de/DE/Themen/Eisenbahnunternehmen/eisenbahnunternehmen_node.html (cit. on p. 23).

[Eis20a] Eisenbahn-Bundesamt. *Bericht des Eisenbahn-Bundesamts*. 2020 (cit. on p. 21).

[Eis20b] Eisenbahn-Bundesamt Sachgebiet 333. *Zusammenstellung der Bestimmungen der Eisenbahn-Signalordnung 1959 (ESO 1959), einschließlich der gemäß ESO (4) genehmigten Signale mit vorübergehender Gültigkeit und der gemäß ESO (5) erlassenen Anweisungen zur Durchführung der ESO, gültig für das Netz der Eisenbahnen des Bundes (EdB)*. 2020 (cit. on p. 21).

[FDR20] J. Falk, F. Dürr, K. Rothermel. "Time-Triggered Traffic Planning for Data Networks with Conflict Graphs". In: *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 2020, pp. 124–136. DOI: 10.1109/RTAS48715.2020.00-12 (cit. on p. 15).

[FGD+21] J. Falk, H. Geppert, F. Dürr, S. Bhowmik, K. Rothermel. *Dynamic QoS-Aware Traffic Planning for Time-Triggered Flows with Conflict Graphs*. _eprint: 2105.01988. 2021 (cit. on pp. 3, 4, 9, 10, 15–17, 41, 49).

[geo] geopy contributors. *GeoPy*. URL: https://github.com/geopy/geopy (cit. on p. 27).

[Gov21a] Government of Germany. *Allgemeines Eisenbahngesetz (AEG)*. 2021 (cit. on p. 23).

[Gov21b] Government of Germany. *Eisenbahnregulierungsgesetz (ERegG)*. 2021 (cit. on p. 23).

[Her06] T. Herrmann. "Stability of timetables and train routings through station regions". In: 2006 (cit. on pp. 15, 17, 37).

[Kar13] C. Karney. "Algorithms for geodesics". In: *Journal of Geodesy* 87 (June 2013), pp. 43–55. DOI: 10.1007/s00190-012-0578-z (cit. on p. 27).

[LLER11] R. Lusby, J. Larsen, M. Ehrgott, D. Ryan. "Railway track allocation: Models and methods". In: *OR Spectrum* 33 (Oct. 2011), pp. 843–883. DOI: 10.1007/s00291-009-0189-0 (cit. on pp. 9, 15, 44).

[Mar] Martin Raifer. *Overpass Turbo*. URL: https://overpass-turbo.eu (cit. on p. 33).

[Mob21] D.-G. for Mobility {and} Transport (European Commission). *EU transport in figures: statistical pocketbook 2021*. LU: Publications Office of the European Union, 2021. ISBN: 978-92-76-40101-8. URL: https://data.europa.eu/doi/10.2832/27610 (visited on 11/10/2021) (cit. on p. 9).

[mof20] mofair e.V., NEE e.V. *Wettbewerber-Report Eisenbahnen*. 2020 (cit. on p. 23).

[Opea] OpenStreetMap Foundation. *OpenStreetMap Foundation*. URL: https://wiki.osmfoundation.org/wiki/Main_Page (cit. on p. 25).

[Opeb] OpenStreetMap Wiki. *OpenStreetMap Wiki*. URL: https://wiki.openstreetmap.org (cit. on p. 26).

[OUF18] H. OUFERROUKH. *European Rail Traffic Management System (ERTMS)*. ERA. Sept. 10, 2018. URL: https://www.era.europa.eu/activities/european-rail-traffic-management-system-ertms_en (visited on 12/03/2021) (cit. on p. 29).

[Pac21] J. Pachl. "Regelung und Sicherung der Zugfolge". In: *Systemtechnik des Schienenverkehrs: Bahnbetrieb planen, steuern und sichern*. Wiesbaden: Springer Fachmedien Wiesbaden, 2021, pp. 39–104. ISBN: 978-3-658-31165-0. DOI: 10.1007/978-3-658-31165-0_3. URL: https://doi.org/10.1007/978-3-658-31165-0_3 (cit. on pp. 19–21).

[PGRR21] L. Philipp, L. Ganter, M. Richter, L. Rönsch. *Use-case oriented Extraction and Processing of Open Street Map Data*. 2021 (cit. on pp. 10, 25).

[RV09] J. Riezebos, W. Van Wezel. "k-Shortest routing of trains on shunting yards". In: *OR spectrum* 31.4 (2009). Publisher: Springer, p. 745 (cit. on p. 16).

[Sar] Sarah Hoffmann. *PyOsmium*. URL: https://osmcode.org/pyosmium (cit. on p. 26).

[SCO18] W. Steiner, S. S. Craciunas, R. S. Oliver. "Traffic Planning for Time-Sensitive Communication". In: *IEEE Communications Standards Magazine* 2.2 (June 2018). Conference Name: IEEE Communications Standards Magazine, pp. 42–47. ISSN: 2471-2833. DOI: 10.1109/MCOMSTD.2018.1700055 (cit. on p. 15).

[SEF+21] M. Salerno, Y. E-Martín, R. Fuentetaja, A. Gragera, A. Pozanco, D. Borrajo. "Train Route Planning as a Multi-agent Path Finding Problem". In: *Advances in Artificial Intelligence*. Ed. by E. Alba, G. Luque, F. Chicano, C. Cotta, D. Camacho, M. Ojeda-Aciego, S. Montes, A. Troncoso, J. Riquelme, R. Gil-Merino. Cham: Springer International Publishing, 2021, pp. 237–246. ISBN: 978-3-030-85713-4 (cit. on pp. 9, 16).

[SPD+16] M. Samà, P. Pellegrini, A. D'Ariano, J. Rodriguez, D. Pacciarelli. "Ant colony optimization for the real-time train routing selection problem". In: *Transportation Research Part B: Methodological* 85 (2016), pp. 89–108. ISSN: 0191-2615. DOI: https://doi.org/10.1016/j.trb.2016.01.005. URL: https://www.sciencedirect.com/science/article/pii/S0191261515301077 (cit. on p. 16).

[Uni] United Nations Economic Commission for Europe. *Total length of railway lines - Statistical Database - United Nations Economic Commission for Europe*. URL: https://w3.unece.org/PXWeb/en/CountryRanking?IndicatorCode=42 (visited on 11/10/2021) (cit. on p. 9).

[Yen71] J. Y. Yen. "Finding the K Shortest Loopless Paths in a Network". In: *Management Science* 17.11 (1971). Publisher: INFORMS, pp. 712–716. ISSN: 00251909, 15265501. URL: http://www.jstor.org/stable/2629312 (cit. on pp. 10, 16).

[YHlL11] H. Yin, B. Han, D. li, F. Lu. "Modeling and Application of Urban Rail Transit Network for Path Finding Problem". In: 124 (Jan. 2011). ISBN: 978-3-642-25657-8, pp. 689–695. DOI: 10.1007/978-3-642-25658-5_81 (cit. on p. 16).

[ZKR+96] P. Zwaneveld, L. Kroon, H. Romeijn, M. Salomon, S. Dauzère-Pérès, S. Hoesel, H. Ambergen. "Routing Trains Through Railway Stations: Model Formulation and Algorithms". In: *Transportation Science* 30 (Aug. 1996), pp. 181–194. DOI: 10.1287/trsc.30.3.181 (cit. on pp. 9, 16, 17).

[ZT16] W. Zhou, H. Teng. "Simultaneous passenger train routing and timetabling using an efficient train-based Lagrangian relaxation decomposition". In: *Transportation Research Part B: Methodological* 94 (2016), pp. 409–439. ISSN: 0191-2615. DOI: https://doi.org/10.1016/j.trb.2016.10.010. URL: https://www.sciencedirect.com/science/article/pii/S0191261516302144 (cit. on p. 16).

All links were last followed on December 6, 2021.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature