Universität Karlsruhe (TH)
Research University • founded 1825

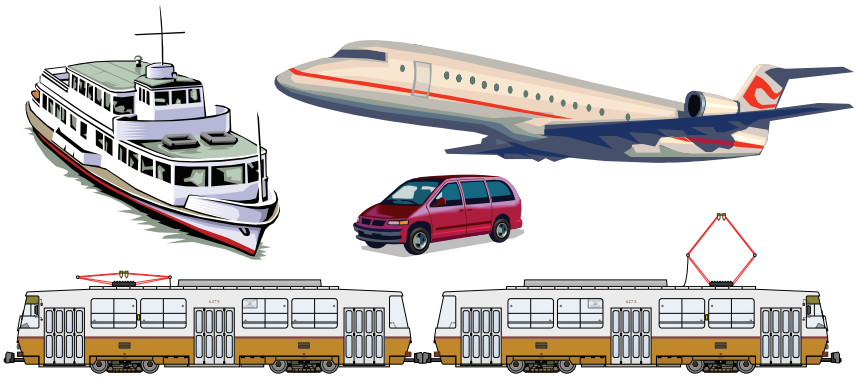
Institut für Theoretische Informatik
Algorithmik I • Prof. Dr. Dorothea Wagner

Diplomarbeit

Multi-Modal Route Planning

Thomas Pajor

30th March 2009



Supervisors:

Prof. Dr. Dorothea Wagner
Institut für Theoretische Informatik,
Universität Karlsruhe (TH),
Germany

Prof. Dr. Christos D. Zaroliagis
Dpt. of Computer Engineering & Informatics,
University of Patras,
Greece

Dr. Daniel Delling
Institut für Theoretische Informatik,
Universität Karlsruhe (TH),
Germany

Dr. Martin Holzer
Institut für Theoretische Informatik,
Universität Karlsruhe (TH),
Germany

The clip arts of the airplane, the ferry boat and the car are taken from the Microsoft Clip Art Gallery. The *Tatra T5C5* Tram from Budapest is drawn by myself.

Contents

1. Introduction **1**

1.1. Related Work 2
1.2. Our Contributions 4
1.3. Overview 6

2. Foundations **9**

2.1. Graph Theory 9
2.2. Languages and Automata 11

3. Models **13**

3.1. Time-Independent and Time-Dependent Models 13
3.1.1. Time-Independent Models 14
3.1.2. Time-Dependent Models 14
3.2. The Road Network 16
3.3. The Railway Network 17
3.3.1. Timetables 18
3.3.2. The Condensed Model 18
3.3.3. Time-Expanded Models 19
3.3.4. Time-Dependent Models 21
3.4. The Flight Network 26
3.4.1. Timetables 26
3.4.2. Using the Railway Model 27
3.4.3. A Flexible Model for Flight Networks 29
3.5. Combining the Networks 33
3.5.1. The Nearest Neighbor Problem 33
3.5.2. Merging and Linking 36

3.6. Summary . . . 38

4. Routing **41**
4.1. The Earliest Arrival Problem and Shortest Paths . . . 42
4.1.1. The Earliest Arrival Problem . . . 42
4.1.2. Shortest Paths . . . 43
4.2. Uni-Modal Routing . . . 45
4.2.1. Time-Independent Routing . . . 46
4.2.2. Time-Dependent Routing . . . 47
4.3. Multi-Modal Routing . . . 51
4.3.1. The Label Constrained Shortest Path Problem . . . 52
4.3.2. Algorithms . . . 53
4.4. Summary . . . 58

5. Speed-Up Techniques **61**
5.1. Basic Ingredients . . . 63
5.1.1. Bi-Directional Search . . . 63
5.1.2. A* with Landmarks (ALT) . . . 66
5.1.3. Arc-Flags . . . 69
5.1.4. Contraction . . . 76
5.2. Core-Based Routing . . . 82
5.2.1. Preprocessing . . . 83
5.2.2. Query . . . 86
5.2.3. Proof of Correctness . . . 87
5.2.4. Discussion . . . 90
5.3. Core-ALT . . . 91
5.3.1. Preprocessing . . . 92
5.3.2. Query . . . 92
5.3.3. Proof of Correctness . . . 94
5.3.4. Discussion . . . 94
5.4. Access-Node Routing . . . 95
5.4.1. Motivation . . . 97
5.4.2. Formal Introduction . . . 98
5.4.3. Preprocessing . . . 100
5.4.4. Query . . . 105
5.4.5. Core-Based Access-Node Routing . . . 107
5.4.6. Proof of Correctness . . . 110
5.4.7. Discussion . . . 113

5.5. Summary 114

6. Experiments **117**
6.1. Input 117
6.1.1. Graphs 118
6.1.2. Automata 120
6.2. Experimental Setup 121
6.3. Multi-Modal Routing 122
6.4. Core-ALT 123
6.5. Access-Node Routing 126
6.6. Summary 130

7. Conclusion **133**
7.1. Future Work 136

A. Data Structures **139**
A.1. Graphs 139
A.1.1. Static Graphs 140
A.1.2. Dynamic Graphs 142
A.2. Piecewise Linear Functions 144
A.3. Finite Automata 145
A.4. Priority Queue 147

B. Raw Data Processing **149**
B.1. Road Data 149
B.2. Railway Data 150
B.3. Flight Data 151

Chapter 1

Introduction

In modern life, route planning is gaining more and more importance. As transportation networks become more complex and mobility in our society more important, the demand for efficient methods in route planning increases even further. Handheld sat-nav systems for cars already established as commodity items and most railway and flight companies offer some kind of route planning facilities through the Internet.

However, all these systems have one common limitation: They only allow route planning within their respective domain. For example, sat-nav system only show the way around the road network. If we rather like to use means of public transportation, we are stuck with the task of getting to and from the nearest station or airport since route planning facilities of public transportation companies usually do not involve the road network. Even worse, when planning a flight, this usually involves putting the following stages of the journey together manually. First, select a convenient flight, after that, choose a train that arrives at the airport in time and, further, match the time of leaving the house correctly in order to catch the train at the station.

The problem of route planning involving different modes of transportation is called *multi-modal route planning*. Our goal is very simple. We would like to be able to state source and target addresses (in the road network) together with a departure time, select our desired means of transportation (for example, car, trains and flights, but the car only in the beginning) and the system should return an optimal route with respect to travel time that shows us what roads to use and which trains and flights to take.

The involved networks may have very different properties. For example, the road network can be used at any given time, however, trains and flights only operate at defined times according to some timetable. The challenge of combining the individual networks in a realistic way such that we are able to perform efficient queries is the topic of this thesis.

1.1. Related Work

Routing is a widely researched topic in computer science, mainly because of its relevance to real world applications. The problem of route planning can be modeled by finding a shortest path on a directed weighted graph. First algorithms to solve this problem are quite old and are presented in the 1950s and 60s by Dijkstra [Dij59], Bellmann and Ford [Jr.56, Bel58] and Hart, Nilsson and Raphael [HNR68] (A*).

While these algorithms compute optimal shortest paths with optimal theoretic time complexity, they are too slow to process real world data sets like large scale road networks, even on today's computers. However, only in recent years computer hardware has become efficient enough to allow the handling of large networks like they occur in route planning, at all. Thus, during the past years research focused on developing *speed-up techniques* to accelerate the basic shortest paths algorithms by reducing their search space. A comprehensive overview is given in [DSSW09a]. In this section we mention work on the subject of road and public transportation networks separately.

Routing in Road Networks. Although, the first attempts to speed up Dijkstra's algorithm were conducted regarding timetable information on railway networks in 1999 (see [SWW99]), in 2005 huge road networks were made publicly available which led research toward road networks. This culminated in the 9th DIMACS Challenge on shortest paths [DGJ09] in 2006.

It turns out that there are a few basic ingredients to all modern speed-up techniques [Sch08a, Del09a]: Bi-directional search, goal-directed search and contraction. In Section 5.1 we present these basic ingredients in more detail.

In short, *bi-directional search* (due to [Dan62, GH05]) not only computes the shortest path from the source s to the target t, but simultaneously computes the shortest path from t to s on the backward graph. The final shortest path is then combined from partial paths obtained by the forward and backward searches. While this approach works well in time-independent networks where the edge weights are constant in the graph, adapting bi-directional routing to time-dependent networks where the edge weights are time-dependent functions is not straightforward [NDLS08].

Regarding *goal-directed search* basically two approaches exist. Based on the A* algorithm by Nilsson and Raphael [HNR68], Goldberg et al. enhanced A* by introducing landmarks to compute feasible potential functions using the triangle inequality [GH05, GW05]. Their approach is called *ALT* and turns out as a very robust technique [BDW07]. The adaption to time-dependency is easy as shown in [DW07].

The second goal-directed approach is using edge-labels to guide the search. Wagner et al. introduced in [WW03, WWZ05] a method called geometric containers where

each edge contains a label that represents some geometric object containing all nodes to which a shortest path begins at the respective edge. During the query, edges that do not contain the target node can be pruned. This approach is refined by Lauther in [Lau04] called *Arc-Flags*. Instead of geometric containers, the graph is partitioned into k cells and k edge-labels (arc-flags) are attached to every edge to indicate whether the respective edge is important for at least one target node in each cell. Regarding the choice of the partition, studies have been conducted using grid based approaches [Lau04, KMS05] and several partitioning algorithms that do not rely on geometric information [MS04, Kar07, Pel07].

Regarding contraction there are a variety of approaches. *Highway Hierarchies* by Sanders and Schultes [SS05, SS06a] exploits the implicitly given hierarchy in road networks regarding different road categories. *Contraction Hierarchies* presented by Geisberger et al. [Gei08, GSSD08] is solely based on contracting the graph yielding a very efficient speed-up technique.

Although, not a basic ingredients as mentioned above, there is another approach for speeding up shortest path queries: Table-lookups. In *Transit-Node Routing* [SS06b, BFM+07, BFSS07] a set of relevant transit nodes are determined where almost all 'far' shortest path pass through. Distances are then precomputed between each pair of transit-nodes. Several approaches exist for selecting transit-nodes: Separators [Mül06, DHM+09], border nodes of partitions [BFM+07, BFSS07, BFM09] and nodes turning out important by other speed-up techniques [BFM+07, BFSS07, SS09].

Several speed-up techniques arose that use combinations of the ingredients from above [HSW04, HSWW06, Sch08a, BDS+08]. Bi-directional routing can be combined with goal-direction and contraction [BDS+08, DN08]. Relevant for this work is the combination of contraction with goal-directed search (especially ALT). In [BDS+08] Core-ALT is introduced for time-independent and time-dependent networks where the graph is contracted yielding a much smaller core-graph on which then ALT is applied. Further combinations of contraction with ALT are presented in [GKW06b, GKW06a, GKW09, DSSW09b]. Combining contraction with Arc-Flags yields a very efficient uni-directional speed-up technique called SHARC [BD09, Del09b] which can be combined further with bi-directional routing. The fastest speed-up techniques on road networks as of today are CHASE (Contraction Hierarchies plus Arc-Flags) and the combination of Transit-Node Routing with Arc-Flags [BDS+08, BDS+09] yielding query times down to 2 microseconds on continental-sized road networks.

Routing in Public Transportation Networks. Routing in public transportation networks turns out harder than in road networks. Research focused on timetable information for railway networks. In [SWW99, PSWZ04, Sch05, PSWZ07, MSWZ07, DPW08]

different models regarding different levels of realism and algorithms are presented. Basically there are two approaches to cope with the inherent time-dependency of a railway timetable: The time-expanded and the time-dependent approach. While the first allows more flexible modeling of additional constraints, the latter yields smaller graph sizes and, thus, faster query times [PSWZ07]. See also Section 3.3 for an in-depth presentation of the several models.

Dijkstra's algorithm can be adapted to solve the Earliest Arrival Problem (i.e., we minimize travel time) on both the time-independent and time-dependent approach. Experimental studies on timetable information have been conducted using speed-up techniques from road networks [PSWZ07, BDW07, BDW09, DPW08]. It turns out that speed-up techniques are much harder to adapt to timetable information than one might expect, although a combination of ALT and Arc-Flags harmonizes well [DPW08].

Furthermore, in public transportation networks multi-criteria optimization turns out more important (e.g., not only minimize the travel time, but also costs, transfers, etc). This problem has been studied by Müller-Hannemann et al. [MW01, MS07, DMS08]. However, in this work we restrict ourselves to single-criteria search, i.e., optimizing travel time alone.

Multi-Modal Routing. Multi-modal route planning can be solved by the Label Constrained Shortest Path Problem [BBH+09]. In [BJM00] an extensive theoretical study is conducted on the complexity of the Label Constrained Shortest Path Problem regarding various types of languages. It is shown in [BJM00] that the Label Constrained Shortest Path Problem is solvable in deterministic polynomial time when restricted to regular languages by a generalization of Dijkstra's algorithm [Dij59]. The first experimental study is conducted in [BBJ+02] using the special case of linear regular languages.

Basic speed-up techniques including bi-directional search (see [Dan62, GH05]), A* (see [HNR68]) and the Sedgewick-Vitter-Heuristic (see [SV86]) are tested in the context of multi-modal routing in [Holo8, BBH+09].

1.2. Our Contributions

In this thesis we approach the problem of efficient multi-modal routing on heterogeneous continental-sized networks composed of road, railway and flight networks. Basically, our work is divided into three parts: Modeling, Routing and Speed-Up Techniques.

Models. The first part is devoted to introducing realistic models for each of the network types. While regarding road and railway networks we use existing models, we work out that despite similarities between railway and flight timetables, using the existing railway models on flight timetables yields unnecessarily large graphs. Thus, we propose a series of new flight models with increasing complexity (and realism) that resemble flight timetables in a more fitting manner. Moreover, we present efficient methods for linking each of the networks together in order to obtain a huge multi-modal graph.

Routing. In this part we lay foundations for multi-modal route planning by augmenting DIJKSTRA's algorithm first to time-dependency and later to multi-modality.

Speed-Up Techniques. Our main contributions stem from this part of our work. Query times using naive algorithms turn out too high as in our experiments we observe query times of up to 87 s on our largest inputs. Thus, we systematically develop methods for accelerating the shortest path search.

First, we identify basic ingredients that are common to all modern speed-up techniques. We adapt each of the ingredients to multi-modality or show that adaption turns out hard.

From the ingredients we then choose contraction and develop a multi-modal Core-Based Routing approach upon it. The basic idea is to contract the input graph to a much smaller core graph. By using contraction only on the time-independent road network we obtain a feasible contraction based technique that is general in the sense that the query still works with all 'reasonable' language based constraints. Furthermore, our Core-Based Routing approach allows any speed-up technique to be applied on the core.

In previous work it was shown that contraction harmonizes well with goal-direction. Unfortunately, regarding Arc-Flags we work out that this goal-directed technique is very hard to adapt to multi-modality in a meaningful way as paths pruned during preprocessing may turn out important if we change the language constraints after preprocessing. Thus, we adapt the ALT algorithm to multi-modality and combine it with our Core-Based Routing approach yielding a robust multi-modal variant of Core-ALT.

However, experiments on Core-ALT only yield mild speed-ups. Therefore, we pursue the following observation. The biggest parts of our multi-modal graphs are made up by the time-independent road network. However, computing shortest paths on pure road networks is easy as with recent speed-up techniques this can be done in a matter of microseconds. The public transportation networks are the hard part. Un-

fortunately, in multi-modal routing the hardness of the time-dependent public transportation network is carried over to the road network.

Thus, we 'outsource' the road network by precomputing for each node of the road network a set of access-nodes in the public transportation network that are at least important once during the day. This set of important access-nodes is relatively small. A query is then conducted by jumping directly into the public transportation network from the source and target nodes of the road network, skipping the search through the road network. By these means, we reduce a point-to-point query on the whole graph to a many-to-many query on the proportionally small public transportation network. Local queries that do not use the public transportation network (but solely the road network) can be computed in parallel in a few microseconds time by using existing uni-modal speed-up techniques on road networks. This carries almost no weight regarding overall query time.

Hence, we separate the road network from the public transportation network in a modular way such that we are able to apply different algorithms on the road and public transportation networks orthogonally. We call this approach *Access-Node Routing* and we are able to achieve speed-ups of up to over 30 000 even when using plain Dijkstra on the public transportation network.

Even further, Access-Node Routing can be combined with our multi-modal Core-Based Routing approach, yielding a Core-Based variant of Access-Node Routing that drastically reduces the amount of preprocessed data whereas the loss in query performance is almost negligible. With this approach we are able to perform intercontinental queries on a graph containing the road networks of Europe and North America together with a flight network in 2.3 ms time.

Please note that we do not intend to develop new speed-up techniques solely for road or public transportation networks in this work. Instead, with our main contribution Access-Node Routing we present a *method* that allows isolating the public transportation network from the road network in a multi-modal context. Thus, the two main parts of our multi-modal networks (road and public transportation networks) with their very different properties can be treated individually.

1.3. Overview

This thesis is organized as follows.

Chapter 2: Foundations. Chapter 2 lays the essential foundations for our work. We define basic notion for graphs including (time-dependent) edge weights and paths. Furthermore, we introduce the concept of languages (whereas we focus on regular

languages) and non-deterministic finite automata which correspond to regular languages.

Chapter 3: Models. In Chapter 3 we present models for each of our network types. First, we distinguish between time-dependent and time-independent models on a more abstract level. Regarding time-dependency, we introduce the term of piecewise linear functions which are used as travel time functions in all of our time-dependent models in this work. Right after that, we conduct a survey on available models for road and railway networks working out their individual pros and cons with respect to multi-modal routing. While for both the road and railway networks we make use of existing models (the road model is time-independent while we use the realistic time-dependent railway model with constant transfer times), we work out in Section 3.4 why using the railway model for flight networks is undesirable. Instead, we propose a set of models with increasing complexity to model flight timetables more adequately yielding smaller graph sizes. Finally, in Section 3.5 we describe how individual uni-modal graphs from each of the networks (road, rail and flight) are merged into a huge multi-modal graph. It turns out that we have to solve large instances of the Nearest Neighbor Problem which requires some extra considerations.

Chapter 4: Routing. This chapter is devoted to introducing basic routing. First of all, in Section 4.1 we formally state the Earliest Arrival Problem and show how it is related to the Shortest Path Problem on our graphs. We then introduce Dijkstra's algorithm, which can compute shortest paths on uni-modal networks in Section 4.2. We start off with the simple time-independent case and generalize the algorithm to cope for time-dependency. Furthermore, regarding time-dependency we present algorithms for both solving time queries where we are interested in a shortest path for one given departure time τ and also profile queries where we are interested for a travel time function that represents shortest paths for every possible departure time τ during the day.

The second part of this chapter introduces algorithms for multi-modal route planning in Section 4.3. In the beginning, we work out that the simple Shortest Path Problem is not sufficient for computing adequate routes on multi-modal networks, as the shortest path may correspond to an undesirable sequence of different modes of transportation (e.g., using the car in the middle of a journey may be undesirable). Thus, we augment the Shortest Path Problem yielding the Label Constrained Shortest Path Problem which restricts valid shortest paths through languages. Finally, in Section 4.3.2 we present a generalization of Dijkstra's algorithm that efficiently solves the Label Constrained Shortest Path Problem on multi-modal

graphs restricted by regular languages.

Chapter 5: Speed-Up Techniques. This Chapter makes up the biggest part of this thesis. As most of today's high-performance speed-up techniques are composed of a few basic ingredients, in Section 5.1 we give a brief overview about the most important ingredients bi-directional routing, goal-directed search (ALT and Arc-Flags) and contraction. For every ingredient we follow the same paradigm: We explain each ingredient with regard to uni-modal time-independent routing, as this is the scenario they were initially developed for. Then we show how they can be adapted to time-dependency and finally to multi-modality explaining the difficulties arising along the way.

With the basics laid out, in Section 5.2 we introduce Core-Based Routing tailored to multi-modal route planning. Its basic idea is to restrict contraction to the time-independent road network. In Section 5.3 we build upon Core-Based Routing by introducing the ALT algorithm on the core graph obtained from the contraction routine. This speed-up technique is called Core-ALT. Finally, we introduce Access-Node Routing in Section 5.4 which computes for each node in the road network a set of relevant entry- and exit-points into the public transportation network, thus, allowing the restriction of the query to the public transportation network. Regarding each of the speed-up techniques we present the preprocessing and the query algorithm in separate subsections. Furthermore, we give detailed proofs of correctness.

Chapter 6: Experiments. In Chapter 6 we conduct several multi-modal experiments. At first, in Section 6.1 we present our inputs used throughout the experiments (graphs and automata). In Sections 6.3, 6.4 and 6.5 we show figures for basic multi-modal routing, Core-ALT and Access-Node Routing, respectively. While Core-ALT yields mild speed-ups, with Access-Node Routing we are able to perform queries on continental-sized networks in 1.9 ms time corresponding to a speed-up of over 30 000 when compared to the basic multi-modal query algorithm.

Chapter 7: Conclusion. Chapter 7 gives a final conclusion and an outlook regarding future research on the topic of multi-modal route planning.

Appendices. We provide two appendices regarding implementation details. In Appendix A we introduce the most relevant data structures used for our experiments. Especially, we present our static and dynamic graphs, our finite automata and our implementation of piecewise linear functions. In Appendix B we describe the process of converting the raw data on which our road, railway and flight graphs are based upon.

Chapter 2

Foundations

In this section we develop the basic notation which is needed throughout the work. Since all of our algorithms work on graphs, the underlying concepts of graph theory are introduced first. Furthermore, for multi-modal route planning we require basic concepts of regular languages and automata.

2.1. Graph Theory

Graphs. A *graph* is a tuple $G = (V, E)$ consisting of a finite set V of *nodes* and a set $E \subseteq V \times V$ of *edges*. We say there is an edge from $u \in V$ to $v \in V$ if and only if $(u, v) \in E$. All of our graphs are *directed*, i.e., the direction of an edge is important. A reflecting edge $e = (v, v)$ is called a *loop*. The graph obtained by flipping all edges is called the *backward graph* $\overleftarrow{G} := (V, \overleftarrow{E})$ where $(u, v) \in \overleftarrow{E} \Leftrightarrow (v, u) \in E$.

A *node induced subgraph* $G' \subseteq G$ with $G' = (V', E')$ and $V' \subseteq V$ is obtained by $E' := \{(u, v) \mid u \in V', v \in V' \text{ and } (u, v) \in E\}$. Further, an *edge induced subgraph* $G' \subseteq G$ given $E' \subseteq E$ is obtained by the node set $V' := \{v \mid \exists (u, v) \in E' \text{ or } (v, u) \in E', u \in V\}$.

Edge Weights. The main difference between time-independent and time-dependent route planning is the type of edge weights. Whereas for time-independent route planning it is sufficient to have constant weights, we generalize this concept to periodic functions to accommodate for different edge weights at different times of day.

All functions associated with the edges are elements of a function space $\mathfrak{F}$ consisting of functions $f : \mathbb{R}_0^+ \to \mathbb{R}_0^+$. The function associated to an edge $e \in E$ is denoted by f_e. In this work, we restrict ourselves to *periodic* functions with a (time-) *period* Π, meaning that for all $\tau \in \mathbb{R}_0^+$ it has to hold that $f(\tau) = f(\tau \mod \Pi)$. Furthermore, we are only interested in *travel time functions*, which means that the function value $f(\tau)$ is a time

span. For that reason, all functions have to fulfill the *FIFO-property*, i.e., for any two $\tau_1, \tau_2 \in \mathbb{R}_0^+$ with $\tau_1 < \tau_2$ it must hold that $f(\tau_1) + \tau_1 < f(\tau_2) + \tau_2$. The FIFO-property guarantees that along an edge it is never possible to depart later but arrive earlier. If it holds for all $\tau \in \mathbb{R}_0^+$ that $f(\tau) = c$ for some $c \in \mathbb{R}_0^+$ then f is called a *constant function*.

Let $f, g \in \mathfrak{F}$ be two functions, then we define $f \oplus g := f + g \circ (f + \mathrm{id})$ as *link-operation*. Here, id denotes the identity function. If both f and g are constant functions we shall simply write $f + g$. This operation is used to cascade functions, i.e., to compute the travel time along two or more subsequent edges. Please note that linking is neither associative nor commutative, hence, the order of evaluation is important. The *minimum operation* of two functions $f, g \in \mathfrak{F}$, denoted by $\min(f, g)$, is obtained by taking the minimum value from f and g for each input $\tau \in \mathbb{R}_0^+$. Sometimes, we also refer to this operation as *merge-operation* since it merges two functions together.

The lower bound of a function f is the minimum value of f for any $\tau \in \mathbb{R}_0^+$, denoted by $\underline{f} := \min_{\tau \in \mathbb{R}_0^+} f(\tau)$. Analogously, the upper bound of a function is the maximum value of all its values: $\overline{f} := \max_{\tau \in \mathbb{R}_0^+} f(\tau)$.

A time-independent lower/upper bound graph $\underline{G}/\overline{G}$ can be obtained from G by replacing each edge function with their lower/upper bounds.

Paths. A *path* P in G is a sequence of nodes $[v_1, v_2, \ldots, v_k]$ such that for each $1 \leq i < k$ the condition $(v_i, v_{i+1}) \in E$ holds. If additionally $v_1 = v_k$, then we call P a *cycle*. Note that a path may contain certain nodes multiple times without being a cycle. A *subpath* $S \subset P$ is a path itself which is fully contained in P. By $|P|$ we denote the *number of edges* along the path.

The *length* of a path P is the sum of its edge weights along the path and is denoted by

$$\begin{aligned} \mathrm{len}(P) &:= \sum_{i=1}^{k-1} f_{(v_i, v_i+1)} \\ &= f_{(v_1,v_2)} \oplus f_{(v_2,v_3)} \oplus \cdots \oplus f_{(v_{k-1},v_k)}. \end{aligned} \tag{2.1}$$

Note that $\mathrm{len}(P)$ yields a function which can be interpreted as the travel time along the path at any given time point τ. However, if all edge weights along the path are constant, then $\mathrm{len}\, P$ is constant as well.

Often we are only interested in the length of a path for a given departure time $\tau \in \mathbb{R}_0^+$. In this case $\mathrm{len}(P, \tau)$ is defined recursively as follows.

- For $|P| = 1$: $\mathrm{len}(P, \tau) := f_{(v_1,v_2)}(\tau)$,
- for $|P| > 1$: $\mathrm{len}(P, \tau) := \mathrm{len}(P - v_k, \tau) \; + \; f_{(v_{k-1},v_k)}(\tau + \mathrm{len}(P - v_k, \tau))$.

In other words, $\text{len}(P, \tau)$ is obtained by walking along the path starting at time τ and evaluating each edge function at time τ plus the distance already covered along the path. For that reason, the result obtained by $\text{len}(P, \tau)$ is a scalar value.

The *distance* between two nodes $u, v \in V$, written by $\text{dist}(u, v, \tau)$, for a given departure time τ, is the minimal length of all paths P from u to v. Note that it is possible that there might be more than one minimal path from u to v. A minimal path P between two nodes u and v at time τ is called *shortest path* from u to v. The travel time function representing the length of all shortest path over all times of day is denoted by $\text{dist}^*(u, v)$.

We call two nodes $u, v \in V$ *connected*, if there exists a path from u to v. If this is true for all pairs of nodes $u, v \in V$, we call the whole graph connected. For a non-connected graph G, a connected subgraph $G' \subseteq G$ is called *strong connected component* of G.

2.2. Languages and Automata

Languages. Let Σ be a finite set of symbols often called *alphabet*. A sequence $w := [\sigma_1, \sigma_2, \ldots, \sigma_k]$ of symbols from Σ is called a *word*. For simplicity, we just write $w = \sigma_1\sigma_2 \ldots \sigma_k$. The *length* of a word is the number of symbols it is composed of. The *empty word* is denoted by ε and has length 0. For two words $w_1 := \sigma_1 \ldots \sigma_k$ and $w_2 := \sigma_{k+1} \ldots \sigma_l$ the *concatenation* of the two words $w := w_1 w_2$ is obtained by simply appending the second word to the first, hence $w = \sigma_1 \ldots \sigma_k \ldots \sigma_l$.

A not necessarily finite set L of words over Σ is called a *language* over Σ. All operations on sets like union, intersection and difference also apply to languages. If L is an arbitrary language, then the i'th power set of L is defined recursively by

- If $i = 0$: $L^0 := \{\varepsilon\}$ and
- if $i > 0$: $L^i := \{wv \mid w \in L^{i-1} \text{ and } v \in L\}$.

With this notion, we introduce an additional operation. The *Kleene-Closure* of a language L is defined by

$$L^* := \bigcup_{i \geq 0} L^i. \tag{2.2}$$

In the special case of $L = \Sigma$, the Kleene-Closure yields all possible words (including the empty word) that can be created by the alphabet of Σ. Finally, for two languages $L_1, L_2 \subseteq \Sigma^*$ the *concatenated language* $L_1 \cdot L_2$ is obtained by $L_1 \cdot L_2 := \{vw \mid v \in L_1 \text{ and } w \in L_2\}$.

It turns out that for our purposes general languages are too powerful. Therefore, we restrict ourselves to regular languages for which we give a detailed definition.

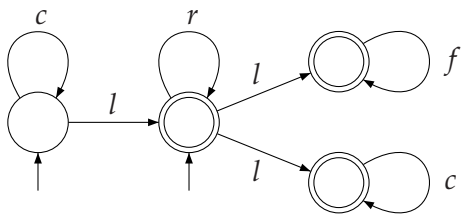


Figure 2.1.: A simple finite automaton given by its transition graph with 4 states from which 2 are initial states and 3 are final states.

Definition 1 (Regular Languages)**.** *Let Σ be an alphabet. Then a language L over Σ is regular if and only if it conforms to the following construction rules.*

- *The empty language $\emptyset$ is regular.*
- *For each $\sigma \in \Sigma$ the singleton language $\{\sigma\}$ is regular.*
- *If L_1 and L_2 are regular languages, then $L_1 \cup L_2$, $L_1 \cdot L_2$ and L_1^* are also regular languages.*

Besides using regular expressions, another way to describe regular languages is by defining a finite automaton.

Finite Automata. A *non-deterministic finite automaton* $\mathcal{A} := (Q, \Sigma, \delta, S, F)$ consists of a finite set Q of states, an alphabet Σ, the transition function $\delta : Q \times \Sigma \to \mathcal{P}(Q)$, a set S of initial states and a set F of final states. Most of the time, we describe a finite automaton visually by its transition graph: States $q \in Q$ are drawn as nodes and for each state $q \in Q$ and every symbol $\sigma \in \Sigma$ we draw an edge from q to q' labeled by σ if and only if $q' \in \delta(q, \sigma)$. Initial states are marked by an incoming edge-tip whereas final states are twin-framed. Figure 2.1 shows a simple example of a finite automaton.

Let $L \subseteq \Sigma^*$ be an arbitrary language. We say that a word $w \in L$ is *accepted* by $\mathcal{A}$, if there is a path in the transition graph starting at an initial state $q_0 \in S$, leading to a final state $q_f \in F$ and where the subsequent edges on the path are labeled by the subsequent symbols of w. If no path fulfilling these properties exists, the word is *rejected*. If it holds for every word $w \in L$ that w is accepted by $\mathcal{A}$, we say that the language L is accepted by $\mathcal{A}$.

By Kleene's Theorem [Kle, RS59] each regular language L can be described by a (non-deterministic) finite automaton $\mathcal{A}$ in the sense that for every word $w \in \Sigma^*$ it holds that $\mathcal{A}$ accepts w if and only if $w \in L$. On the other hand it is true that for every finite automaton $\mathcal{A}$ the set of words accepted by $\mathcal{A}$ has the properties of being a regular language. Hence, the terms regular language and finite automaton can be interchanged.

Chapter 3

Models

In this chapter, we introduce the different models we use for our multi-modal routing algorithms. Our work is mainly based on three different types of networks: Road networks, railway networks and flight networks. All of these networks have quite different properties which makes it unsuitable to use the same approach for all three of them. Hence, we introduce the model behind each of the network types in a separate section. Whereas the models for the road and railway networks are already well known, we show that—despite strong similarities between rail and flight timetables—it is not adequate to use the same modeling approach for flights as we do for railways. For that reason, we propose a new approach for flight networks that is more compact while still maintaining the required flexibility and realism.

To compute actual multi-modal queries, the query algorithm has to utilize multiple networks simultaneously. For that reason, the different models—which yield different graphs—have to be 'glued' together. Simply speaking, we do this by using the x and y coordinates attached to the nodes and solve the Nearest Neighbor Problem on a subset of pairs of nodes. The process of combining the networks is described in the last section of this chapter.

3.1. Time-Independent and Time-Dependent Models

Before we turn toward each of the network types, we point out the fundamental difference between time-independent and time-dependent models, since both versions are used in this work. While we use the time-independent approach for the road network, we use time-dependent models for the railway and flight networks. The main difference is due to the assigned edge-weights which are constant in one case and functions in the other. Moreover, shortest paths in time-dependent networks depend

on the departure time at the source.

3.1.1. Time-Independent Models

Time-independent routing has been studied in great detail [DSSW09a] and is mostly used in road networks. In a time-independent network each edge e is assigned one constant value $w(e)$ which may be travel time, geographical distance or any other metric that we like to minimize. Suppose we have an edge reflecting a road segment in a road network. If we assign this edge a constant value, our query will always produce the same result when using this edge. While this is perfectly sufficient for certain metrics like geographical distance (The geographical distance of a road segment does not depend on the time), it might not be realistic enough for others. If we use travel time as metric, imagine using a motorway during rush hour in contrast to off hours. We are much slower on the same motorway during the rush hour due to traffic congestions.

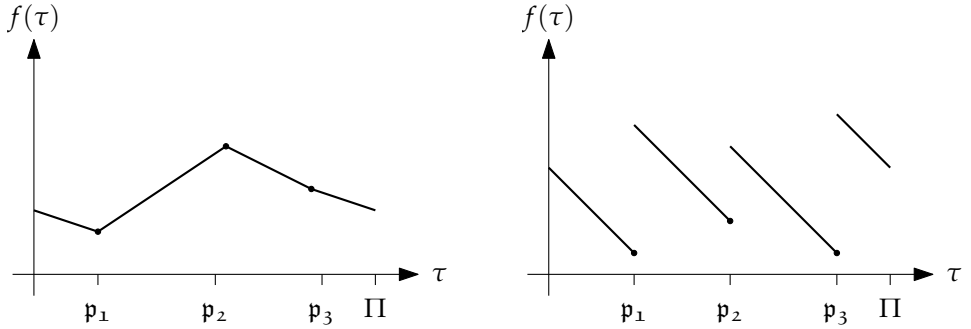
Not having time-dependencies makes the model simpler. To solve shortest path queries on such a network, we can use Dijkstra's algorithm with only minor modifications (See also Section 4.2.1). For that reason lots of research focused toward accelerating shortest path queries on time-independent models (mostly road networks) which resulted in very impressive speed-up techniques. An overview is given in [DSSW09a].

While in a road network the absence of time-dependency still yields useful queries, other models like those for railway timetables most certainly do not. The problem of time-dependency is inherent to these models, because trains only operate at certain times and therefore the choice of the quickest route highly depends on the departure time of the journey. Although, there are approaches for eliminating the time-dependency in the model, namely by time-expansion (See Section 3.3), the more canonical approach is to generalize the time-independent model itself to incorporate the concept of time-dependency.

3.1.2. Time-Dependent Models

In time-dependent routing we do no longer have constant weights assigned to the edges. To accommodate for time-dependency, we replace the edge weights by arbitrary functions f from some function space $\mathfrak{F}$. The shortest s-t-path in a time-dependent model then depends on the departure time τ_s of the source node. This might result in shortest paths of different length for different departure times or—in general—even a completely different route.

In Section 2.1 we restricted ourselves to periodic functions $f : \mathbb{R}_0^+ \to \mathbb{R}_0^+$. However, using arbitrary periodic functions is not adequate for an efficient implementation of the query algorithm.



(a) Road type with variable gradient.

(b) Public transportation type with fixed gradient.

Figure 3.1.: Two piecewise linear functions. The left one is typical for time-dependent road networks where we linearly interpolate between two interpolation points. The function to the right is typical for public transportation networks.

Piecewise Linear Functions. A periodic function $f : \mathbb{R}_0^+ \to \mathbb{R}_0^+$ is called *piecewise linear* if it consists of a finite number of segments of linear functions. Let f be a piecewise linear function then f can be described by a finite set $\mathfrak{P}$ of interpolation points where each interpolation point $\mathfrak{p}_i \in \mathfrak{P}$ consists of a departure time τ_i and an associated function value $f(\tau_i)$.

The value of f for an arbitrary time τ is then computed by interpolation. This is done differently for time-dependent road networks and public transportation networks. Whereas in road networks we interpolate linearly between two subsequent interpolation points, the travel time function along a public transportation edge is interpreted as follows. First we have to wait for the next train or airplane to depart and then we have to add its mere travel time along that edge to that. Hence, for some arbitrary time point τ we use the nearest interpolation point τ_i in the future and interpolate by the formula

$$f(\tau) = -\gamma \cdot (\tau_i - \tau) + f(\tau_i). \tag{3.1}$$

Here, γ denotes the *fixed* gradient of f. In order to fulfill the FIFO-property (cf. Sections 2.1 and 3.3.4 on page 25), $\gamma \in [-1, 0]$ has to hold.

Note that our functions are periodic with time-period Π. For that reason τ_i is chosen with respect to the periodicity of f, meaning that if no τ_i exists with $\tau_i > \tau$, then τ_i is set to be the smallest interpolation point τ_0 of f. The function space consisting of all piecewise linear functions with a certain gradient γ is denoted by $\mathfrak{F}_L(\gamma)$. Without proof we like to mention that the function space $\mathfrak{F}_L(\gamma)$ for an arbitrary value of γ is closed under addition as well as both link- and merge-operations. Figure 3.1 illustrates

two examples of piecewise linear functions that are typical for time-dependent route planning models.

With the differences between time-independent and time-dependent route planning laid out, we now turn toward describing each of the models in more detail.

3.2. The Road Network

The road network is the simplest model of all the three because its graph representation is straightforward. Junctions are modeled as nodes and an edge $e = (u, v)$ between two junctions $u, v \in V$ is inserted if and only if a road segment from u to v exists in the road network. Note that if in reality the road between u and v is a two-way road, then two edges (u, v) and (v, u) are inserted into the graph as well.

Because we set ourselves the goal of using very large scale road networks for our experiments, i.e., the road network of whole Europe and North America (See Section 6.1), using the time-dependent approach on the road network is not viable. The space (and therefore memory) required to store all interpolation points for every edge-function in the network is simply too high. Note that the memory consumption of the multi-modal query algorithm is higher than it is the case for an uni-modal algorithm (cf. Section 4.3.2) which also adds to the space we require. Furthermore, in [Delo9b, Delo9a] it has been shown that both link- and merge-operations on road type edge functions are very expensive, as the number of interpolation points may increase up to their sum when merging two road functions. This consumes even more space and computational time. So, in order to keep the problem manageable, we decide on using the time-independent approach for our road network, i.e., only constants are assigned to the edges.

Edge Weights. The edge weights $w(e)$ in the road network represent the average travel time on the specific road segment. The average travel time is computed by taking the average traffic speed on the specific road segment, which is then charged against the geographical length of that segment to obtain the travel time.

Foot Edges. When planning a multi-modal route using, it is a realistic assumption not having a car available everywhere along the journey. In order to still be able to make point to point queries in the road network, we thus require some sort of foot edges, so any stage of the journey can be covered by foot. From a theoretical point of view, we simply insert additional edges between two junctions u and v if the road segment is available to pedestrians. The edge weight of the foot edge is then computed by taking the geographical length of the road segment and assuming an

Figure 3.2.: Example of a road network graph showing the center of Karlsruhe.

average walking speed s of a pedestrian to obtain an average travel time. Note that there might be additional nodes created that are not reachable by car, for example in pedestrian precincts. Just as well, there might be nodes in the graph that are not reachable by foot, like those belonging to motorways.

While the approach of modeling foot edges by creating multi-edges in the graph seems a smart thing to do, in our implementation we stick to only using one edge per road segment. Instead, we assign two separate weights to the edges. In regard to our graph data structure, this has the advantages of consuming significantly less space (See Appendix A.1.1). Furthermore, we do not need to support multi-edges in our data structures and algorithms.

Figure 3.2 shows as a tiny example of a road network: The center of Karlsruhe.

3.3. The Railway Network

While the model of the road network is quite canonical, there are several approaches for modeling railway networks. After formally introducing timetables—which are the essence behind any of the railway models—we give an overview about existing models discussing their pros and cons and finally go into more detail about the realistic time-dependent model which we actually use in this work.

Be reminded that in the road network we use travel time as metric. So, in order to be consistent with the road network all railway models presented here also use travel time as metric.

3.3.1. Timetables

The basis for each of the models is a *timetable* from which we construct some kind of graph on which we can then compute shortest paths.

A traffic timetable is a tuple $(\mathcal{C}, \mathcal{B}, \mathcal{Z}, \Pi)$ where $\mathcal{B}$ is a set of stations, $\mathcal{Z}$ a set of trains, Π the periodicity of the timetable and $\mathcal{C}$ a set of elementary connections. An *elementary connection* from $\mathcal{C}$ is defined by a tuple $c := (Z, S_1, S_2, \tau_1, \tau_2)$ and is interpreted as train $Z \in \mathcal{Z}$ going from station $S_1 \in \mathcal{B}$ to station $S_2 \in \mathcal{B}$, departing at S_1 at time $\tau_1 < \Pi$ and arriving at $\tau_2 < \Pi$. Please note that for one elementary connection the train has to go from S_1 to S_2 without stops in-between. So, a train going along some route will consist of multiple elementary connections in the timetable.

Some attention has to be put at computing the the travel time of an elementary connection c. If the arrival time τ_2 is greater than the departure time τ_1 then the travel time of c is simply the difference $\tau_2 - \tau_1$. However, due to the periodicity of the timetable it is also possible to depart in the evening and arrive during the next day. In this $\tau_2 < \tau_1$ holds and the travel time is composed of the travel time from τ_1 until midnight plus the travel time from midnight until τ_1. With regard to the time period Π we therefore obtain for the travel time

$$\Delta(\tau_1, \tau_2) := \begin{cases} \tau_2 - \tau_1 & \text{if } \tau_2 \geq \tau_1, \\ \Pi - \tau_1 + \tau_2 & \text{otherwise.} \end{cases} \tag{3.2}$$

Note that the Δ function can be used to compute arbitrary travel times between two points in time τ_1 and τ_2.

In the subsequent sections we give a brief overview of the existing railway models

3.3.2. The Condensed Model

The most basic approach is the time-independent condensed model. For every station $S \in \mathcal{B}$ of the timetable there is exactly one node $v \in V$ in the graph. An edge $e = (u, v)$ is introduced, if and only if at least one elementary connection exists in the timetable that goes from u to v. The edge weight of an edge $e = (u, v)$ is set to

$$w(e) := \min_{\substack{c \in \mathcal{C}, \\ u = S_1(c), \\ v = S_2(c)}} \Delta(\tau_1(c), \tau_2(c)), \tag{3.3}$$

the minimum travel time of all connections going from u to v.

Discussion. While the condensed model yields a very small graph and represents the structure of the railway network adequately, a shortest path query only results in a

lower bound regarding the travel time between two stations. So, for computing exact travel times this model cannot be used and, therefore, we discard it in our work.

3.3.3. Time-Expanded Models

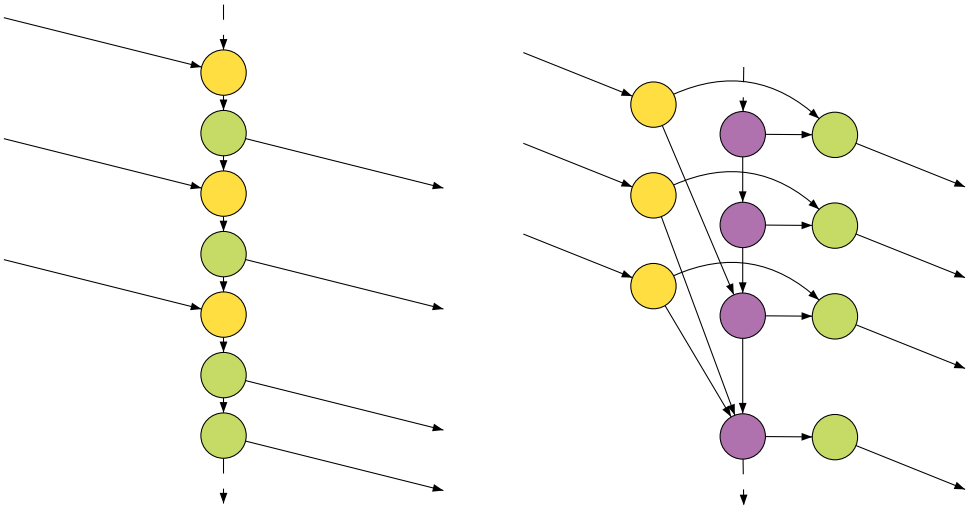
The reason why the condensed model is not useful for exact shortest path queries is that it does not account for departure and arrival times in the timetable. To address this issue while still being able to use the time-independent approach, the time-expanded model was developed. Historically there have been two versions: The simple time-expanded model [Sch05] is a direct mapping from an itinerary as we defined it above. Shortest path queries yield correct results. However, the simple model does not account for realistic transfers. When switching trains at a station S during the journey, there is most likely a minimum transfer time transfer(S) involved that one needs to get from one train to another. To incorporate this aspect, the simple model has been enhanced to the realistic time-expanded model.

Simple Version. Nodes in the graph no longer correspond to stations in the timetable, but rather to *events*. In the simple version of the time-expanded model there are two types of events: Departure events and arrival events. For each elementary connection $c = (Z, S_1, S_2, \tau_1, \tau_2)$ there is a departure event of train Z at station S_1 and time τ_1 and an arrival event of train Z at station S_2 and time τ_2. So, we basically insert two nodes u and v into the graph which correspond to the departure and arrival events. To keep track to which station a node belongs, each node is assigned its station S and furthermore its timestamp τ when the event occurs.

Besides the nodes, two types of edges are inserted. For two events (nodes) belonging to the same elementary connection c there is an edge from the departure to the respective arrival event. The edge weight is set to be the travel time $\Delta(\tau_1(c), \tau_2(c))$. In order to allow transfers, nodes belonging to the same station S are sorted in ascending order wrt. their timestamp. For two subsequent nodes v_i, v_{i+1} having timestamps τ_i and τ_{i+1} there is an internal station edge $e := (v_i, v_{i+1})$ with weight $\Delta(\tau_i, \tau_{i+1})$. Finally, to allow transfers over midnight, there is an edge from the latest node v_k to the earliest node v_0 with weight $\Delta(v_k, v_o)$. Figure 3.3a gives a small example of the model.

As discussed above, the simple version of the time-expanded model allows arbitrary small transfers. The realistic time-expanded model overcomes this issue.

Realistic Version. To cope with realistic transfers, the timetable $\mathcal{C}$ is extended by a function transfer : $\mathcal{B} \to \mathcal{N}$ which assigns each station in the timetable a transfer time transfer(S).



(a) Simple time-expanded model. **(b)** Realistic time-expanded model.

Figure 3.3.: The same station modeled in both the simple and the realistic time-expanded railway models. Arrival nodes are yellow, departure nodes green and transfer nodes purple. While in the simple version of the model the second train can be reached from the first, this is not possible in the realistic model.

Regarding the realistic version of the time-expanded model there are now three types of nodes: transfer, departure and arrival nodes. While arrival nodes still represent arrival events of the timetable, departure events are modeled by a pair of nodes consisting of a transfer and a departure node. For each elementary connection $c \in \mathcal{C}$ we insert three nodes $v_{c,\text{tra}}, v_{c,\text{dep}}, v_{c,\text{arr}}$. The stations and timestamps are assigned to the nodes analogously to the simple version of this model, whereas the transfer and departure nodes have the same values.

Regarding edges, there are now five different edges.

(1) Departure-Edges.
For each elementary connection their transfer and departure nodes are connected by an edge $(v_{c,\text{tra}}, v_{c,\text{dep}})$ with weight 0.

(2) Connection-Edges.
For each elementary connection there is an edge $(v_{c,\text{dep}}, v_{c,\text{arr}})$ connecting the departure node with the arrival node.

(3) Station-Edges.
For every station S only their transfer nodes are sorted in ascending order and internal station edges are inserted in the same manner as in the simple model.

(4) Transfer-Edges.
These edges effectuate the demand of upholding the minimum transfer time criterion. Let S be a fixed station. For each arrival node v_{arr} of S at time timestamp(v_{arr}) we look for the smallest transfer node v_{tra} at S satisfying $\Delta(v_{\text{arr}}, v_{\text{tra}}) \geq \text{transfer}(S)$. An edge is then inserted accordingly.

(5) Train-Edges.
In order to be able to stay in the same train, for each arrival node v_{arr} we insert an edge to the unique departure node v_{dep} belonging to the same train. If there is no such departure node (the train could end at the current station), this step is omitted.

Figure 3.3 gives an example of the realistic time-expanded model and compares it to the simple version of the model.

Discussion. The time-expanded model 'rolls out' the time-dependencies of the railway timetable and allows exact shortest path queries wrt. the Earliest Arrival Problem (introduced in Section 4.1). For a given departure time τ at some source station S the source node is determined to be the earliest station or transfer node s (depending on the version of the model used) with timestamp$(s) \geq \tau$. While the greatest advantage of the time-expanded model is its easy adaption of the standard Dijkstra query algorithm, there are numerous disadvantages. First of all, the target node is not known in advance of the query, since the arrival time is unknown. While this is no problem to correctness, it makes the use of bidirectional speed-up techniques hard. Furthermore, the size of the graph gets extremely large consuming a lot of memory and also leading to a very big search space of Dijkstra's algorithm. Although it could be shown recently that the realistic time-expanded model can be enhanced with the effect that the search space is reduced [DPW08], both disadvantages (not being able to use bidirectional routing and the high memory consumptions) remain. Therefore, we decide against using the time-expanded model in our work.

3.3.4. Time-Dependent Models

This section covers the time-dependent approach for modeling railway timetables. It overcomes all disadvantages of the time-expanded model, but with the penalty of introducing time-dependency in the graph. Historically—as with the time-expanded approach—there were two versions of the model developed. The simple version, again, does not respect the minimum transfer time criterion, whereas the realistic version deals with this issue by enhancing the simple model.

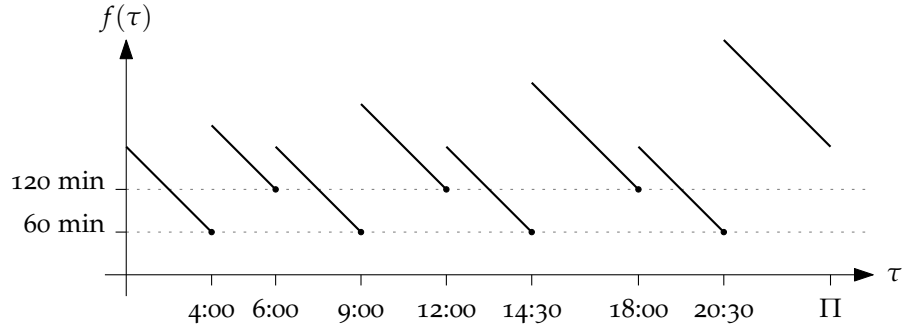


Figure 3.4.: This is a rail piecewise linear function with 7 interpolation points. There are 4 fast trains departing at 4:00, 9:00, 14:30 and 20:30, each of which needs 60 minutes for the journey. The slow trains at 6:00, 12:00 and 18:00 take 120 minutes for their journey. Edge weights between times of departure are filled up with waiting time for the next departing train.

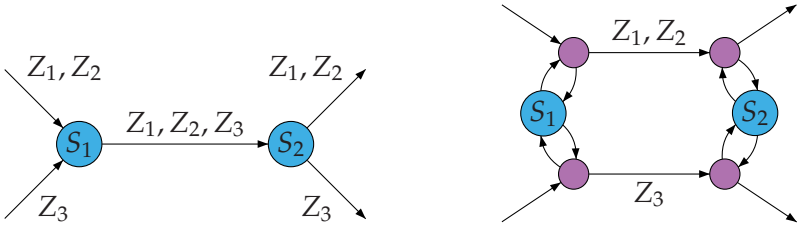
Simple Version. The simple version is an immediate augmentation of the condensed model introduced before. Again, the node set is exactly the set of stations and a *connection edge* between two nodes u and v is inserted if and only if there is at least one connection from u to v in the timetable. However, instead of using lower bounds as edge weights, the edges become time-dependent.

As edge function type we use piecewise linear functions as introduced in Section 3.1.2. For each connection $c = (Z, S_1, S_2, \tau_1, \tau_2)$ in the timetable we add an interpolation point $\mathfrak{p} := (\tau_1, \Delta(\tau_1, \tau_2))$ to the function f that belongs to the edge between S_1 and S_2. We can imagine this as a correspondence of interpolation points to departure events on the particular edge of the network. So, if we evaluate the function f at one of its interpolation points τ_i, the value of $f(\tau_i)$ results precisely in the travel time of the i'th train on that segment. If we, on the other hand, evaluate the edge at an earlier point $\tau < \tau_i$, we have to wait at S_1 for the train to depart. Therefore, the edge weight $f(\tau)$ is composed of the travel time $f(\tau_i)$ plus the waiting time. This yields the equation

$$f(\tau) = \underbrace{(\tau_i - \tau)}_{\text{waiting time}} + \underbrace{f(\tau_i)}_{\text{travel time}} . \tag{3.4}$$

Figure 3.4 shows a typical railway piecewise linear function while Figure 3.5a shows a small example of the simple time-dependent model.

Again, the simple model does not account for realistic transfer times. Therefore, the simple model has been enhanced to a realistic version coping with this issue.



(a) Simple model. **(b)** Realistic model.

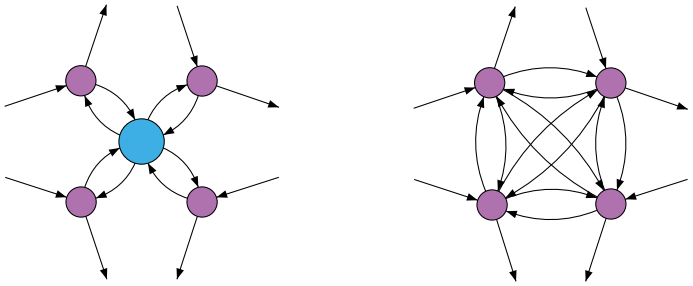
Figure 3.5.: This is an excerpt of two stations S_1 and S_2 of the time-dependent model. The simple version, i.e., without realistic transfers, is shown to the left and the realistic version is shown to the right. Station nodes are blue, whereas route nodes are drawn in purple. The trains Z_1 and Z_2 are equivalent (use the same route), while train Z_3 uses a different route. In the realistic version of the model switching between trains of different routes can only be done by going through the station node. The transfer time is then charged on the edges from station to route nodes.

Realistic Version. As with the realistic time-expanded model, we extend the timetable by a function transfer : $\mathcal{B} \to \mathbb{R}_0^+$ in order to assign each station a fixed transfer time. The graph of the realistic version of the time-dependent model is then constructed as follows.

For each station $S \in \mathcal{B}$ we introduce a super-node called *station node* into the graph. For simplicity, if we write S and refer to a node, we always refer to the station node of S. However, these station nodes are not directly interconnected as in the simple model. The basic idea is to have an additional node type called *route nodes* in the graph and have trains that take the same route go through subsequent route nodes. These route nodes are then connected to their respective station nodes with the proper weight matching the transfer time.

More formally, we divide the set of trains $\mathcal{Z}$ into *train routes*. The set of train routes is denoted by $\mathcal{R}$. Each train route $R \in \mathcal{R}$ is a maximal subset of $\mathcal{Z}$ containing only trains following the exact same sequence of stations $[S_1, S_2, \ldots, S_k]$. In other words, we can regard two trains Z_1 and Z_2 as equivalent, denoted by $Z_1 \sim Z_2$, if they follow the same route. Hence, the set $\mathcal{R}$ of routes contains the equivalence classes regarding the equivalence relation $\sim$ on $\mathcal{Z}$.

Now let $[S_1, S_2, \ldots, S_k]$ be the sequence of stations belonging to some train route $R \in \mathcal{R}$. Then for each station $S_i \in R$ we insert a route node r_i into the graph. Furthermore, we connect subsequent route nodes with time-dependent edges $e = (r_i, r_{i+1})$. The interpolation points of the function f_e at e are created just as with the simple model: For each elementary connection c that belongs to a train Z using the route R we insert an interpolation $\mathfrak{p} = (\tau_1, \Delta(\tau_1, \tau_2))$. The fixed gradient of all functions is again -1.

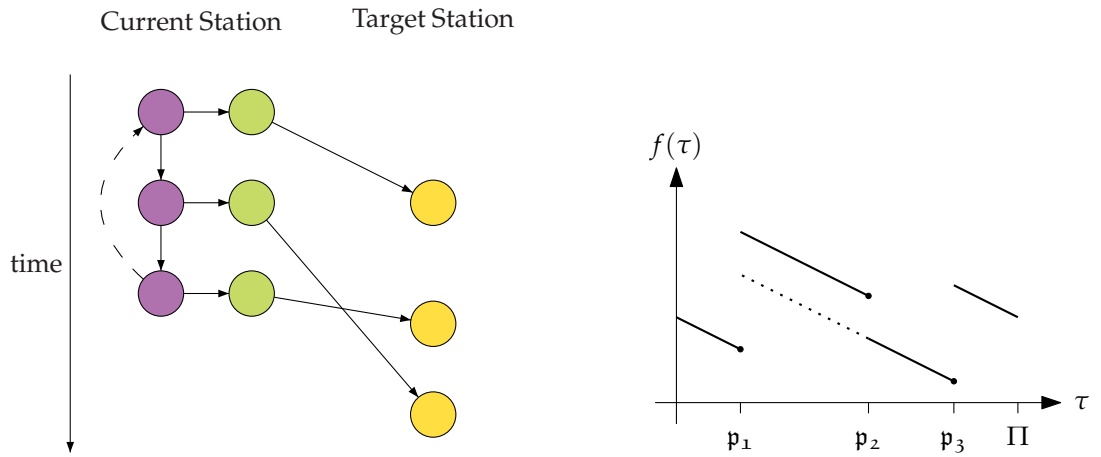


(a) Constant transfer times. **(b)** Variable transfer times.

Figure 3.6.: This is a comparison between constant and variable transfer times in the realistic time-dependent model within a station. With variable transfer times, the number of edges grows quadratic in the number of route nodes and there are no more station nodes in the graph.

To allow switching trains between different routes, we insert additional *transfer edges* into the graph. For some station S consider each of the route nodes r belonging to S. Then we insert two additional edges: One edge (r, S) from the route node to the station node with constant weight 0. This models getting off a train, which is not charged with any time. Another edge (S, r) is inserted which models boarding the train. The weight of this edge is set to transfer(S). This approach is called *constant transfer time* approach, since all trains at a station are boarded by the same value of transfer(S). Figure 3.5b illustrates the realistic version of the time-dependent model along the simple version of this approach.

Variable Transfer Times. The realistic time-dependent model can be generalized even further to account for variable transfer times. This allows modeling of different transfer times between different train routes running through a station S. For example, there might be trains of a route r_1 that are connecting trains of another train route r_2 and therefore always arrive at the same platform. Then the minimum transfer time could be less than in the general case when changing trains requires switching platforms. However, there are a few reasons why we do not use variable transfer times in our work. First, our raw data on which our graphs are based on, only contain constant transfer times and, second, modeling variable transfer times increases the complexity of the graph, particularly regarding the number of edges. But the main refutation is that according to [PSWZ07] the station nodes are no longer present in the graph. This makes linking the railway network to the other networks difficult as we need some kind of unique station nodes to link them to the road network (See Section 3.5.2 for details).



(a) Time-expanded graph. **(b)** Time-dependent edge function.

Figure 3.7.: Violation of the FIFO-property in the time-expanded model (left) and the respective edge function of the time-dependent model (right). The third train overtakes the second. In the time-expanded model this has no effect on the query algorithm. However, in the time-dependent model, for some τ between $\mathfrak{p}_1$ and $\mathfrak{p}_2$, it would pay off to use $\mathfrak{p}_3$ for evaluating f (dashed line) instead of using $\mathfrak{p}_2$.

FIFO-Property. The time-dependent model has one restriction when compared to the time-expanded model. The time-dependent Earliest Arrival Problem becomes $\mathcal{NP}$-hard to solve if the edge weight functions of the graph do not fulfill the FIFO-property, see [OR90]. The FIFO-property states that for each function f it has to hold that $f(\tau_1) + \tau_1 < f(\tau_2) + \tau_2$ for any two $\tau_1 < \tau_2$. If we transfer this onto our model, this means that overtaking of trains must be prohibited, i.e., there must not exist two trains $Z_1, Z_2 \in R$ for which between two subsequent stations S_1 and S_2 train Z_1 leaves after Z_2 at S_1 but arrives before Z_2 at S_2. See Figure 3.7 for an illustration. The figure also shows that the time-expanded model is immune to this problem. However, this problem can be avoided (with a small increase in graph size) by splitting the route R into a minimal set of routes where each of the routes does not contain conflicting trains any longer. Since overtaking trains are very rare in real world timetables, the increase in graph size is insignificant.

Discussion. Time-dependent model allow exact shortest path queries for the Earliest Arrival Problem in a time-dependent fashion. At least for the simple version and the realistic version with constant transfer times, both source and target nodes are known in advance: They are simply the station nodes corresponding to the source and target station of the query. The variable transfer time version requires a many-to-many shortest path query, since each of the route nodes may be a potential source resp. target

node of the query. Furthermore, when using a time-dependent model the graph size is much smaller than that of the time-expanded model. The simple approach has the same graph size as the condensed model which is extremely small, while the graph size of the realistic model increases approximately by a factor of 5 compared to the simple model, which is still much smaller than any of the realistic models.

All these advantages come with a penalty. First, the graph is no longer time-independent. This requires an augmentation of the query algorithm (See Section 4.2.2), as well as additional memory to store the piecewise linear functions. However, experiments revealed [PSWZ07] that these disadvantages generally do not outweigh the advantage of the smaller graph size, since query times on the time-dependent model are smaller. This led us to the decision to use the realistic time-dependent model (with constant transfer times) as railway model in our work.

3.4. The Flight Network

This section covers the model behind the flight networks we use in our work. In contrast to railway models, there has been no explicit development of an efficient model for timetable information so far. Using the same approach as for railways yields graphs with unnecessarily many nodes and edges. Therefore, we propose several new approaches. However, we restrict ourselves to time-dependent versions of the model.

The section is organized as follows. First we introduce flight timetables which form the basis of our models. Next, we work out why the realistic time-dependent railway model is not suited well to model flight timetables. Based on these observations, we develop a family of flight models: The constant-time model, the flight-class model and the variable-time model, each being a generalization of the previous one.

3.4.1. Timetables

Creating a flight graph requires an underlying flight timetable. The flight timetable has almost the same structure as a railway timetable (See Section 3.3.1). However, we are using the terms *airport* instead of station and *flight* instead of train. Therefore, a flight timetable is a tuple $(\mathcal{C}, \mathcal{A}, \mathcal{F}, \zeta, \Pi)$ where $\mathcal{C}$ is a set of elementary (flight-) connections, $\mathcal{A}$ a set of airports, $\mathcal{F}$ a set of flights and Π the time period. An addition to the railway timetable is $\zeta : \mathcal{A} \to \mathbb{Z}$ which maps each airport to the *timezone* it belongs to. Timezones are represented as UTC (coordinated universal time) offset from `UTC+0` with the same resolution as time points in general (e.g., seconds). An elementary connection $c \in \mathcal{C}$ is a tuple $c = (F, A_1, A_2, \tau_1, \tau_2)$. Again, this is interpreted as flight $F \in \mathcal{F}$ departing at airport $A_1 \in \mathcal{A}$ at time τ_1 and arriving at airport $A_2 \in \mathcal{A}$ at time

τ_2. Note that τ_1 and τ_2 are time points relative to the timezone of the airports A_1 and A_2.

Computing Flight Lengths. In order to compute valid flight lengths, we cannot simply use Δ as defined in Section 3.3.1. This would yield false results because we omitted to account for the timezones. So, suppose there is a flight between two airports A_1 and A_2 with departure time τ_1 at A_1 and arrival time τ_2 at A_2. Then the actual flight duration can be computed by converting both the departure and arrival time to `UTC+0` time. This can be done by subtracting the timezone offset off the time, i.e.,

$$\tau_1' := \tau_1 - \zeta(A_1) \mod \Pi \qquad \text{and} \qquad \tau_2' := \tau_2 - \zeta(A_2) \mod \Pi. \tag{3.5}$$

Given these results, we can compute the length of a flight using Δ as usual: $\Delta(\tau_1', \tau_2')$. For the sake of simplicity, we write τ^* when referring to the time point τ after conversion to `UTC-0` time.

3.4.2. Using the Railway Model

Because of the striking similarity between railway and flight timetables it is tempting to use the same modeling approach as for railways. However, it turns out that there are some differences regarding routes and also regarding procedures at airports which makes the use of the railway model unfitting.

Adapting the Simple Time-Dependent Railway Model. The simple time-dependent railway model can be adapted to flight timetables in a straightforward manner. Nodes correspond to airports and a time-dependent edge is inserted between two airports A_1 and A_2 if and only if there is at least one flight departing at A_1 heading to A_2. Interpolation points for the edge functions are created identically as for railway functions. While this may lead to exact solutions to the Earliest Arrival Problem, the problem of instantaneous transfers has an even greater impact on realism for flights than on railway queries. Usually we have to spend more than an hour for procedures at the airport before we actually depart. Therefore, adapting the simple model is of no avail.

Adapting the Realistic Time-Dependent Railway Model. As we worked out for railways, the realistic version of the time-dependent railway model is a good way to incorporate realistic transfer times into the model. This requires us to define a function $\text{transfer} : \mathcal{A} \to \mathbb{R}_0^+$ which maps each airport to some (constant) transfer time. Before we work out why this might not suffice for realism, we show a few other disadvantages that are inherent to the realistic model when applied to flight timetables.

No Routes. There are no routes in flight timetables (in the sense of the realistic railway model), or to say it differently: All routes have length 1. There is no case where we stay in the plane and travel *through* an airport. Even if there are flights in the timetable that have stops in between, these are always regarded as direct connections between the first and the last airport.

This makes the concept of routes in the graph somewhat awkward, but if we wanted to force this concept on us, it would have the following consequence. For each airport A we had one route node per airport where at least one flight reaches to and also one route node per airport where at least one flight arrives from. Basically, the number of route nodes per airport can be bounded by 2 times the number of neighbors of A (in the condensed graph). This leads directly to the second point.

High Number of Neighbors. Whereas in railway networks the number of neighbors for each station is relatively small (less than 5) for most of the stations [DPW08], airports tend to have many more neighbors. So, when mixing this with the previous observation, we end up having extremely many nodes per airport. Even worse, if we consider *variable transfer times* we end up having $\Theta(n^2)$ edges between the route nodes of the airport A, if A has n neighbors. For small n—like in railway networks—this is not such a big problem but in flight networks this weighs heavier.

Constant Transfer Time Assumption is Unrealistic. If we bring back to mind how the transfer time criterion is modeled into the realistic railway model (with constant transfer times), namely that we charge the transfer time for *entering* a train (no matter if we transfer between trains or enter a train the first time of the journey), this is another drawback concerning realism. Usually, if we board a flight at the departure airport we have to spend more time for checking in our luggage and security checks than for switching flights which only requires us to walk from one gate to another. For that reason, modeling realistic airport procedures requires at least two different types of times per airport: *Check-in time* to account for the whole procedure involving check-in, security checks and the waiting time at the gate and, secondly, a *transfer time* which only accounts for the time needed to switch planes within the airport. Another desirable issue having modeled is a third type of time for getting off at the destination airport. This time should cope for customs and baggage claim. While this can be modeled into the realistic railway model by changing the edge weights of the edges connecting route nodes to station nodes, incorporating a dedicated transfer time could only be done by inserting transfer edges between all route nodes. Here, the high number of neighbors per airport leads again to unnecessarily many edges in the graph.

These problems lead us to proposing a family of entirely new models for flight

timetables.

3.4.3. A Flexible Model for Flight Networks

The basis of our flight model is a flight timetable $(\mathcal{C}, \mathcal{A}, \mathcal{F}, \zeta, \Pi)$. Furthermore, we introduce three different time functions to model the various procedures in an airport.

- Check-in time $\mathcal{T}^{\text{check-in}} : \mathcal{A} \to \mathbb{R}_0^+$.
  This accounts for the whole process from arriving at the airport until the departure of the plane composed of checking-in, passing security checks (additionally maybe customs) and also the accounted waiting time at the gate plus the boarding time of the plane.

- Check-out time $\mathcal{T}^{\text{check-out}} : \mathcal{A} \to \mathbb{R}_0^+$.
  This accounts for the reverse process: Leaving the plane, passing customs while leaving the gate area and finally the time required to claim baggage.

- Transfer time $\mathcal{T}^{\text{transfer}} : \mathcal{A} \to \mathbb{R}_0^+$.
  This time accounts for the time transferring between two planes. Usually, this only involves leaving the plane, walking to another gate and boarding the new plane.

Note that we assume that all three time functions do not depend on flights. In favor of more realism, this assumption is weakened in the second and third versions of our model.

Model I: Constant-Time Model. The constant-time model uses the time functions exactly as defined above. For each airport $A \in \mathcal{A}$ we insert a super-node into the graph called *terminal node*. Since all flights either begin or end at our airport (Remember, there are no routes), we insert two more nodes per airport: First, a *departure node* which resembles flight departures at A and second, an *arrival node* to model arrivals.

Edges are created as follows. There are three edges within each airport. A *check-in edge* is inserted from the terminal node to the departure node and its weight is set to $\mathcal{T}^{\text{check-in}}(A)$. A *check-out edge* from the arrival node to the terminal node with weight $\mathcal{T}^{\text{check-out}}(A)$ is inserted and finally a *transfer edge* from the arrival node to the departure node with weight $\mathcal{T}^{\text{transfer}}(A)$ is created.

The actual flights are modeled as *flight edges* from the departure node of airport A_1 to the arrival node of airport A_2 if and only if there is at least one elementary connection from A_1 to A_2 in the timetable. The edge weight is time-dependent and interpolation points are created in the same fashion as for the time-dependent railway

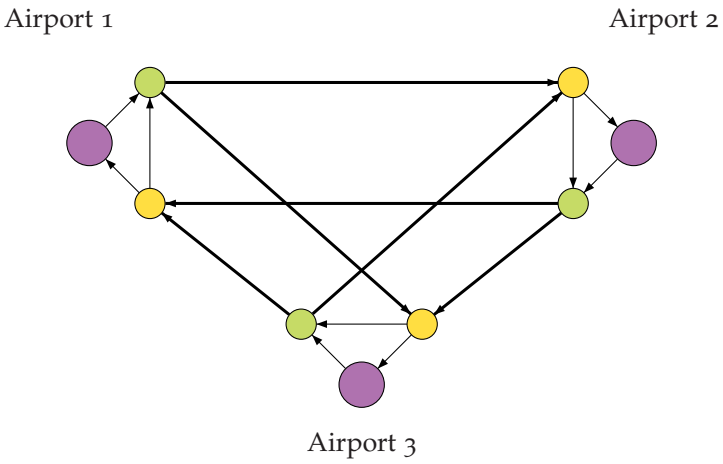


Figure 3.8.: An example of the constant-time flight model with 3 airports. Terminal nodes are purple, departure nodes green and arrival nodes yellow. Bold edges are time-dependent and model flights between the specific airports while the thin time-independent edges allow for check-in, check-out and transfers within the airports.

models. Remember that we have to use τ_1^* and τ_2^* for calculating the length of the flight to account for the different timezones of A_1 and A_2.

An example of the constant-time model is shown in Figure 3.8. While this model yields very small graph sizes its drawback is the assumption that check-in and transfer times are constant for all flights. This is addressed by the flight-class model.

Model II: Flight-Class Model. At first glance, assuming constant times for check-in, check-out and transfers seems reasonable. However, this might not be general enough. Imagine at an airport A having domestic flights and flights abroad. Probably the check-in time is much smaller if we plan on boarding a domestic flight because of heavier security measures and additional customs for flights abroad. This difference also applies to transfers from a domestic flight to a flight abroad and vice versa. For that reason, we augment the definition of $\mathcal{T}^{\text{check-in}}$, $\mathcal{T}^{\text{check-out}}$ and $\mathcal{T}^{\text{transfer}}$ to cope with different flight classes. These augmentations are then incorporated into the flight-class model.

Similarly to the concept of routes in the realistic time-dependent railway model, we partition the set of flights $\mathcal{F}$ into different *flight classes*. The set of flight classes is denoted by $\mathfrak{C}$. The equivalence relation $\sim$ on the set of flights according to which two flights are put into the same class is not predefined. An example might be $F_1 \sim F_2 \Leftrightarrow F_1$ and F_2 are domestic flights, i.e., their source and target airports are both in the same country. With flight classes defined, the time functions are extended as follows. The check-in time function is extended to $\mathcal{T}^{\text{check-in}} : \mathcal{A} \times \mathfrak{C} \to \mathbb{R}_0^+$, the check-out time

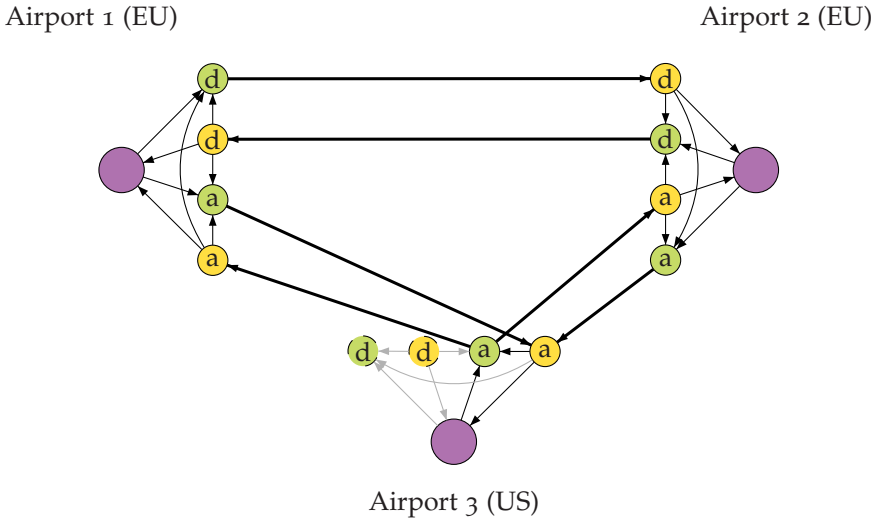


Figure 3.9.: The flight-class model with 3 airports (2 in the European Union and 1 in the United States) and 2 classes, domestic flights (nodes tagged with a 'd') and flights abroad (nodes tagged with an 'a'). The nodes are colored the same way as in Figure 3.8 (departure nodes are green, arrival nodes yellow). The flight edges are always incident to departure and arrival nodes belonging to the same flight class as the flights on the respective edges. Since the US airport has no incoming or outgoing domestic flights, the dashed nodes together with the gray edges would not be created in the real graph.

function is extended to $\mathcal{T}^{\text{check-out}} : \mathcal{A} \times \mathfrak{C} \to \mathbb{R}_0^+$ as well and the transfer-time function is extended to operate on pairs of classes $\mathcal{T}^{\text{transfer}} : \mathcal{A} \times \mathfrak{C} \times \mathfrak{C} \to \mathbb{R}_0^+$ to account for transfers between flights of arbitrary pairs of flight-classes.

To integrate the augmentations of the time-functions into the flight model, the constant-time model is modified as follows. Let $A \in \mathcal{A}$ be an airport. Instead of one departure and arrival node we insert $k = |\mathfrak{C}|$ departure resp. arrival nodes—one for each flight class $\mathfrak{c}_i \in \mathfrak{C}$. The departure and arrival nodes are connected to the terminal node by check-in and check-out edges like in the constant-time model. As edge weights we use $\mathcal{T}^{\text{check-in}}(A, \mathfrak{c}_i)$ and $\mathcal{T}^{\text{check-out}}(A, \mathfrak{c}_i)$ for each of the classes. To model transfers between arbitrary flight-classes, for each pair $\mathfrak{c}_i, \mathfrak{c}_j$ of flight-classes we insert a transfer edge from the arrival node of class $\mathfrak{c}_i$ to the departure node of class $\mathfrak{c}_j$ weighted with $\mathcal{T}^{\text{transfer}}(A, \mathfrak{c}_i, \mathfrak{c}_j)$. Note that this generates $\Theta(k^2)$ edges. Finally, the time-dependent flight edges between two airports A_1 and A_2 are inserted with respect to the correct classes, i.e., if the flight is of class $\mathfrak{c}$, the departure node belonging to $\mathfrak{c}$ at A_1 is used as tail while the arrival node of the same class at A_2 is used as head of the edge. Interpolation points on the functions of the flight edges are created the same way as in the constant-time model while only elementary connections belonging

to the proper flight class are considered.

In order to avoid the creation of unnecessary nodes, at each airport A we can omit the creation of departure and arrival nodes (and their incident edges) which belong to flight classes that do not contain any outgoing resp. incoming connections from/to the airport A.

Figure 3.9 shows a small example of a flight-class model involving two classes: domestic flights and flights abroad. While this version of the flight model is the one we finally choose for our experiments, we provide a further augmentation that supports variable transfer times between arbitrary flights. However, as we work out, the disadvantage regarding graph size, which was an argument against using the railway models in the first place, reoccurs in this approach.

Model III: Variable-Time Model. We introduce the variable-time model by generalizing the flight-class model. Assume that each flight $F \in \mathcal{F}$ is not equivalent to any other flight. Then, the set of flight classes consists of singleton sets containing exactly one flight and it holds that $|\mathfrak{C}| = |\mathcal{F}|$. This already allows us to describe specific check-in, check-out and transfer times for each flight separately by using $\mathcal{T}^{\text{check-in}}$, $\mathcal{T}^{\text{check-out}}$ and $\mathcal{T}^{\text{transfer}}$ due to the definitions from the flight-class model. Therefore, the construction rules of the graph do not need to be modified and can be directly adapted from the flight-class model.

It shall be noted that using this model, the graph may grow fairly large. For each airport A having n neighbors the number of nodes increases to $\Theta(n)$ while the number of edges is even in the magnitude of $\Theta(n^2)$. This is comparable to the realistic time-dependent railway model with variable transfer times.

Discussion. We showed that the realistic time-dependent railway approach is not well suited for designing an efficient flight model. Hence, we introduced a family of three flight models beginning with a simple constant-time model where check-in, check-out and transfer times do not depend on flights at all, to a model incorporating fully variable transfer times for arbitrary flights. While the constant-time approach is too simplistic and the variable-time approach too general, a good trade-off between realism and graph size is the flight-class model. Hence, we use this model in our thesis. However, as flight classes we do not choose domestic flights and flights abroad, but rather different airline alliances (See Appendix B.3). This has two reasons: First, our timetable data is not equipped with the information of flights being domestic or not and, second, our raw timetable data is a combination of timetables of various airline alliances. Furthermore, it is a reasonable assumption to have different transfer times between flights of the same alliance and for flights between different alliances.

3.5. Combining the Networks

In the previous sections we introduced models for each of the networks we use in our multi-modal routing algorithms. However, computing a multi-modal query involves all networks at the same time. Hence, we have to combine the networks into one *multi-modal network*. In this section we assume that we already created the road, railway and flight graphs. To clear things up, we summarize which model we use for each of the network types:

- Road Network.
  We use the time-independent model as introduced in Section 3.2 which is the canonical way to build a road graph. The road graph is denoted by G_{road}.
- Railway Network.
  The realistic time-dependent model with constant transfer times is used as introduced in Section 3.3.4. The graph is denoted by G_{rail}.
- Flight Network.
  For the flight network we use the flight-class model as presented in Section 3.4.3 and it is denoted by G_{flight}.
- Multi-Modal Network.
  After combining the graphs, the resulting network is a multi-modal network. To make multi-modal graphs better distinguishable from uni-modal graphs we typeset multi-modal graphs with bold letters: $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$.

The section is made up as follows. Since combining the graphs requires us to insert links between pairs of nodes of different networks that are geographically next to each other, we introduce the Nearest Neighbor Problem in a theoretical fashion and present an efficient algorithm for solving it on huge datasets. In the subsequent section we then show how to combine the different networks. The combination process can be described by two graph operations: *Merge* and *link* (not to be confused with the merge- and link-operations on functions as presented in Section 2.1) where the merge-operation basically unites two graphs and the link-operation inserts *link edges* to connect them. For the determination for which pairs of nodes link edges should be created, the Nearest Neighbor Problem is used. Finally, we end this chapter with a summary of the main results.

3.5.1. The Nearest Neighbor Problem

In this section we give a formal definition of the Nearest Neighbor Problem and present an algorithm for solving it efficiently on huge instances.

Let $\mathbb{R}^n$ be the n-dimensional vector space over $\mathbb{R}$ and $P \subset \mathbb{R}^n$ a finite set of vectors. The set P is called *candidate points*. Furthermore, let $d : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ be a metric on $\mathbb{R}^n$. The Nearest Neighbor Problem is then defined as follows.

Definition 2 (Nearest Neighbor Problem)**.** *Given a metric space $(\mathbb{R}^n, d)$, a set of candidate points P on $\mathbb{R}^n$ and a set Q of* query points *on $\mathbb{R}^n$ we ask for a map $f : Q \to P$ with the property*

$$f(q) := p \quad \Longleftrightarrow \quad \forall p' \in P : p \neq p' \Rightarrow d(p', q) \geq d(p, q). \tag{3.6}$$

In other words, for each query point $q \in Q$ we try to find the *nearest* candidate point $p \in P$ wrt. the metric d.

We now present two algorithms for solving the Nearest Neighbor Problem. First, we present a linear search algorithm having time complexity $\mathcal{O}(|P|)$ per query. This is the naive approach for solving the problem. However, with respect to our application this turns out to be too slow, since both P and Q are too large. Therefore, we present a clever data structure, called k-d-trees (k dimensional trees) introduced in [Ben75]. k-d-trees are basically an augmentation of binary search trees to k dimensions. Using a k-d-tree on P, the algorithm can be accelerated to answer queries from Q in average time complexity $\mathcal{O}(\log |P|)$.

Linear Search. The naive approach to solve the Nearest Neighbor Problem is to take each query point $q \in Q$ and to scan the list of candidate points P for the the point having minimum distance to q. Basically, this requires the distance computation between each pair $(q, p) \in Q \times P$ of points. This method is called *linear searching* and is illustrated in Algorithm 1.

While the implementation of this algorithm is straightforward, its running time of $\mathcal{O}(|Q| \cdot |P|)$ is not feasible for large sets of P and Q. Our biggest networks yield candidate sets P of up to 50 million points and query sets Q of more than 30 thousand points. This would result in over $150 \cdot 10^9$ distance computations which is too many to be solved in reasonable time.

Using k-d-Trees. k-d-trees, which is an abbreviation for k-dimensional trees, are a data structure introduced by Bentley in [Ben75] specifically designed for geometric search algorithms. We do not give a detailed description of the data structure at this point. The idea is to generalize a binary search tree to k dimensions, which is then able to contain k-dimensional vectors. Queries of k-dimensional points can thus be answered in average logarithmic time. How this data structure can be used to solve the Nearest Neighbor Problem is briefly mentioned in [Ben75]. A more extensive

Algorithm 1: Linear Search

Data: Finite sets of candidate points P and query points Q. A metric d.
Result: A map $f : Q \to P$ assigning each query point its nearest neighbor.

```
1 forall q ∈ Q do
2     mindist ← ∞
3     minpoint ← NULL
4     forall p ∈ P do
5         if d(q, p) ≤ mindist then
6             mindist ← d(q, p)
7             minpoint ← p
8     f(q) ← minpoint
```

tutorial on k-d-trees in conjunction with the Nearest Neighbor Problem is given in [Moo91], while our implementation is based on [Keno4] which is not only able to compute the single nearest neighbor of a query point q, but also the m closest points to q. Our method for solving the Nearest Neighbor Problem is thus reduced to Algorithm 2.

Algorithm 2: k-d-Tree Search

Data: Finite sets of candidate points P and query points Q. A metric d.
Result: A map $f : Q \to P$ assigning each query point its nearest neighbor.

```
1 T ← new k-d-tree
2 T.Build (P)
3 forall q ∈ Q do
4     f(q) ← T.Query (p)
```

The algorithm operates in two phases. First, a k-d-tree is created with all candidate points P. Second, for each query point $q \in Q$ a query on the data structure is stated which yields the nearest neighbor of q. Since each query can be answered in average time $\mathcal{O}(\log |P|)$, the running time of the algorithm reduces to $\mathcal{O}(|Q| \log |P|)$ which takes only a few seconds even on the largest instances of our real world data.

With the theoretical foundations for solving the Nearest Neighbor Problem laid out, we now present our approach on combining each of the networks G_{road}, G_{rail} and G_{flight} into a multi-modal network $\boldsymbol{G}$.

3.5.2. Merging and Linking

In this section we assume that each of the source networks is equipped with functions $\text{coord}_x : V \to \mathbb{R}$ and $\text{coord}_y : V \to \mathbb{R}$ which map every node to its geographical location given in x and y coordinates represented as latitude and longitude values. For our experiment this data was available for every network type.

The process of combining the networks can be described as an application of two operations: The *merge-operation* and the *link-operation*, whereas the link-operation might be applied multiple times. While the merge-operation basically unifies the node and edge sets of multiple graphs, the link-operation takes care of inserting appropriate link edges to connect the networks together.

The Merge-Operation. Given a number of uni-modal graphs $G_1, \ldots, G_n$, each with node and edge sets $G_i = (V_i, E_i)$, the merge-operation yields a *multi-modal graph* $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ where the node and edge sets are simply the union of the node and edge sets of the input graphs, thus $\boldsymbol{V} := V_1 \cup \cdots \cup V_n$ and $\boldsymbol{E} := E_1 \cup \cdots \cup E_n$. To still be able to distinguish between different node types in the resulting graph we introduce a function $\text{label} : V \cup E \to \mathcal{L}_{\text{node}} \cup \mathcal{L}_{\text{edge}}$ which assigns each node/edge a label.

The set of node labels consists of `ROAD_NODE`, `RAIL_NODE` and `FLIGHT_NODE`. The set of edge labels contains `ROAD_EDGE`, `RAIL_EDGE`, `FLIGHT_EDGE`, a new edge type `LINK_EDGE` (we will address later) and additionally the labels `CAR_EDGE` and `FOOT_EDGE`. The reason for having both `ROAD_EDGE` and `CAR_EDGE`/`FOOT_EDGE` as edge labels is the way we store edges in the road graph. In Section 3.2 we worked out that to reduce space consumption, we only store one edge in the road graph for each road segment. To be able to have different car and foot weights, we assign two separate weights to the edges. Therefore, an edge e from the road network results in $\text{label}(e) =$ `ROAD_EDGE`, however, `CAR_EDGE` and `FOOT_EDGE` are still required by our multi-modal query algorithms (cf. Section 4.3.2).

Additionally, we introduce another flag dij, which indicates whether a certain node $v \in \boldsymbol{V}$ can be used as a source resp. target node for Dijkstra's algorithm (or any other shortest path algorithm). While in the road network every node is a legitimate source or target node, in both the railway and flight graphs only station and terminal nodes are used as nodes to instantiate shortest path queries. Hence the dij flag is set to false for all other nodes in the rail and flight networks. Besides, the dij flag will play an important role for the link-operation because link edges are only inserted between nodes having the dij flag set to `true`.

When combining multiple uni-modal graphs, at first the merge-operation is applied on all input graphs. This results in a multi-modal graph $\boldsymbol{G}$. The next step is to

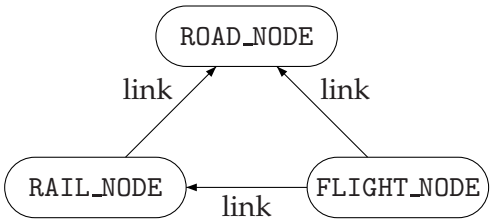


Figure 3.10.: Our approach for linking the different network types together when creating a multi-modal network. Each arrow represents a link-operation between its respective nodes.

repeatedly apply the link-operation which links two subgraphs of different type in $\boldsymbol{G}$ together by inserting link edges.

The Link-Operation. The link-operation involves the process of inserting links between the different networks contained in $\boldsymbol{G}$. This is done through inserting link edges. These links indicate where we are able to switch between networks. One execution of the link-operation only links two networks together. Hence, for a multi-modal network consisting of more than two networks, the link-operation has to be applied multiple times. While in general this can be done in arbitrary ways, the procedure for our specific types of networks is illustrated in Figure 3.10. So, a multi-modal graph consisting of a road, a railway and flight network would require three link-operations: Linking the railway network to the road network, linking the flight network to the railway network and also linking the flight network directly to the road network. We describe the link process using the example of linking a railway network to the road network.

When we link the railway network to the road network, for each station in the railway network our goal is to find a node in the road network that is as close to the station as possible. Then we can insert a link between the two nodes. Finding the nearest road node for a station node is exactly an instance of the Nearest Neighbor Problem derived in Section 3.5.1. The set of candidate nodes P consists of all nodes $v \in \boldsymbol{V}$ with $\mathcal{L}_{\text{node}}(v) = \texttt{ROAD\_NODE}$, whereas the set Q of query points is the set of all station nodes, thus $v \in \boldsymbol{V}$ with $\mathcal{L}_{\text{node}}(v) = \texttt{RAIL\_NODE}$ and $\text{dij}(v) = \texttt{true}$.

More generally, if we want to link a network of type T_1 to a network of type T_2, we define $Q := \{v \in \boldsymbol{V} \mid \mathcal{L}_{\text{node}}(v) = T_1 \text{ and } \text{dij}(v) = \texttt{true}\}$ and equivalently $P := \{v \in \boldsymbol{V} \mid \mathcal{L}_{\text{node}}(v) = T_2 \text{ and } \text{dij}(v) = \texttt{true}\}$. Solving the two dimensional Nearest Neighbor Problem on (P, Q, d) with an appropriate metric d (we explain what we mean by "appropriate" right after this paragraph) yields for each node $q \in Q$ of type T_1 the closest node p of type T_2 from P. Because it is possible that there might be no connection between two nodes in the real world (e.g., not every airport has a railway

station close by), we drop the pair (q, p) if $d(q, p) > d_{\max}$, where $d_{\max}$ is a threshold parameter indicating the maximum distance for which links should still be inserted. Finally, we insert two edges $e_1 = (p, q)$ and $e_2 = (q, p)$ into the graph with both having weight $d(p, q)/\mathfrak{s}$ where $\mathfrak{s}$ denotes the average walking speed of a pedestrian.

Please note that the link-operation is not symmetric. Linking the railway network to the road network implies that for each station one link to the nearest road node is inserted. If we linked the road network to the railway network, we would end up inserting links for each node in the road network (each intersection, for that matter) to the nearest railway station—which is most probably not what we wanted to do. For that reason the direction of the arrows in Figure 3.10 is important.

Distance Metrics. When describing the link-operation we did not commit ourselves to a specific metric d for the nearest neighbor search. Because our x and y coordinates are in geographical form (latitude and longitude), using the Euclidean metric leads to bad results when computing $d(p_1, p_2)$. Hereby, the error in distance increases the more p_1 and p_2 lie apart. The solution to this problem is to use geodetic distances on a solid resembling the form of earth, e.g., the GRS80-ellipsoid [Mor92] which is also used by the Global Positioning System (GPS).

While using the GRS80-ellipsoid directly as metric in the nearest neighbor search would yield the most accurate results, the implementation of the k-d-tree [Ken04] available to us only supports the Euclidean metric. However, as mentioned in Section 3.5.1, we are able to retrieve the m nearest neighbors for a query point q. Because for small distances the error using the Euclidean metric is relatively small, we use a cascaded approach. For each query point q we compute all m nearest neighbors P_q according to the Euclidean metric using a k-d-tree and select the best one from P_q by applying the linear search algorithm (cf. Algorithm 1) on $Q' := \{q\}$ and P_q using the GRS80-metric. For small values of m (we set $m = 100$) this produces results of high quality while still being reasonably fast.

3.6. Summary

In this section we introduced our multi-modal model. First, for each component of the multi-modal network (road network, railway network and flight network), we gave an overview over established models (as far as they existed) and chose the model best suited for our scenario. Furthermore, we showed how these models can be combined in order to obtain a multi-modal network. These are the main results.

- Our road network consists of time-independent car and foot edges, whereby

we decided against using multi-edges for roads that are utilized by both cars and pedestrians. Instead, we use only one edge per road segment but with two distinct weights (car and foot). This allows us saving some space.

- Our railway network is made up of the realistic time-dependent approach with constant transfer times. The reason for deciding against the time-expanded approach is our requirement for bidirectional routing. Not using variable transfer times is mainly due to the fact that the variable transfer times model contains no designated station nodes we can use to link the graph to other networks. Furthermore, our raw timetable data only contains constant transfer times.

- The railway models are not suited for usage as flight model. This is due to the fact that flight timetables differ from railway timetables in the sense that all routes have length 1. Furthermore, procedures at airports differ significantly from procedures at railway stations, which requires us to incorporate different types of transfer times into the flight model. Hence, we developed a new family of time-dependent flight models, from which the flight-class model is used as our model of choice.

- Combining each of the different networks into a huge multi-modal graph involves two operations: Merge and link, while the latter may need repeated execution. In short, the nodes and edges of the graphs are united and link edges are inserted to connect the networks. Furthermore, linking requires an efficient way of solving the Nearest Neighbor Problem. We use a cascaded approach therefor: We compute the m nearest neighbors using the imprecise Euclidean distance metric with a fast algorithm on k-d-trees. For refinement, we then use a linear search algorithm on the result set using the GRS80-ellipsoid for a more precise metric.

Chapter 4

Routing

In this chapter we introduce routing algorithms. We start by giving a formal definition of the Earliest Arrival Problem and show that it is equivalent to the Shortest Path Problem on our graphs. Next, we tend to the simplest case for solving the Shortest Path Problem: Uni-modal time-independent networks. Here, Dijkstra's algorithm [Dij59] yields optimal solutions. This algorithm can be augmented to cope with time-dependent networks [CH66]. We discuss two types of queries on time-dependent networks: *Time queries*, where we are interested in the best route for a fixed departure time τ and *profile queries* where we demand best paths for all possible departure times τ within the time period Π. Profile queries can be computed by a label correcting algorithm that is very similar to Dijkstra's algorithm [Dea99].

We then turn toward multi-modal routing, which is the main part of this chapter. First, we lay some theoretical foundations. We show that using an uni-modal route planning algorithm without modifications on a multi-modal graph can lead to very undesirable paths. For example, a computed shortest path may ask us to take a car between two train connections. For that reason we augment the Shortest Path Problem itself which leads us to the definition of the Label Constrained Shortest Path Problem. A proper definition of the Label Constrained Shortest Path Problem involves the concept of languages introduced in Section 2.2. We summarize some theoretical results about the complexity of the Label Constrained Shortest Path Problem and derive that the Label Constrained Shortest Path Problem can be solved in polynomial time when we restrict ourselves to regular languages and prove this by introducing a further generalization of Dijkstra's algorithm. Its basic idea is to use the product network of the multi-modal graph together with the transition graph of a finite automaton. However, this yields graph sizes which are too large to fit into memory. Hence, in the last part of this chapter, we modify the multi-modal

algorithm in a way that the product network is not required explicitly. Thus, we are able to reduce the amount of required memory significantly.

4.1. The Earliest Arrival Problem and Shortest Paths

When we are planning a journey from s to t, there might be diverse criteria as to which we choose the 'best' route. The first goal coming to mind, is minimizing the travel time, i.e., arriving as early as possible at the target. But there might be many other strategies which are reasonable. For example in a scenario where we plan a trip by car, we could be interested in minimizing the mileage in order to save gas. But the amount of required fuel does not necessarily depend on the mileage alone, but on other things like speed profiles. In a railway scenario we could be interested in different criteria as well: A fast journey, a cheap journey, a journey having as few transfers as possible, a journey involving scenic routes, etc. The options are endless.

To make it even more complicated, we could be interested in a combination of several criteria. For example, when planning a journey by railway, on one hand, we would like to be fast, but on the other hand, we do not want to spend too much money. So besides a fast route we are also asking for a cheap solution (maybe involving special price offers for certain routes). In a multi-modal scenario where we have a car available, we could be interested in a fast route that requires us to use the car as shortly as possible; Or maybe we rather prefer flying than using the train, so we are interested in a fast journey using railways as few times as possible. Hence, the variety is even higher when combining different optimization goals.

Using multiple criteria for optimization in route planning is called *multi-criteria search* [MW01, MS07, DMS08] and is not covered in our work. We restrict ourselves to *single-criteria search* algorithms. Beyond that, we restrict ourselves to optimizing *travel time* alone.

4.1.1. The Earliest Arrival Problem

We now give an informal definition of the Earliest Arrival Problem. The problem definition is deliberately brief in order to be applicable to all network types that were introduced in Chapter 3.

Definition 3 (Earliest Arrival Problem)**.** *Given a time-independent or time-dependent network, source and target points s and t in the network, as well as a departure time $\tau < \Pi$, we ask for a* route *in the network with the following properties.*

(i) The route must start at s,

(ii) the departure time at s is τ,

(iii) the route ends at t and

(iv) the length (travel time) of all other routes satisfying the properties (i)–(iii) must be bigger or at least equal.

The demand (ii) can be dropped, if the underlying network is time-independent.

In other words, from all possible routes in the network from source s to target t (starting at time τ), we seek the route that arrives at t first. The term *route* in the above definition depends on the network type. For example, in road networks a route is simply a sequence of road segments, whereas in rail or flight networks a route is an itinerary that tells us which trains or flights to use and where to transfer between trains/flights. In a multi-modal network, street routes and itineraries are combined.

Itineraries. An *itinerary* is a sequence $\mathcal{I}$ of elementary connections from the railway or flight timetable, such that, for consecutive elementary connections $c_i, c_{i+1} \in \mathcal{I}$ it holds that the arrival station or airport of c_i matches the departure station or airport of c_{i+1}. This yields a chain of elementary connections we can use to travel through the network. A more extensive and formal definition of a railway itinerary is given in [PSWZ07]. The same definition also applies to flight itineraries.

4.1.2. Shortest Paths

Shortest paths are of fundamental interest because of their many applications to real world problems. In our case, the Earliest Arrival Problem can be reduced to the Shortest Path Problem, as we show later in this section. But before we get into that, we give a formal definition of the Shortest Path Problem.

Definition 4 (Shortest Path Problem)**.** *Given a weighted, directed time-independent, time-dependent or mixed graph $G = (V, E)$, a source node $s \in V$, a target node $t \in V$ and a departure time $\tau < \Pi$, we ask for a path $P = [v_1, \ldots, v_k]$ with the following properties.*

(i) The path begins at s, thus $v_1 = s$,

(ii) the path ends at t, thus $v_k = t$ and

(iii) for all paths P' having the properties (i) and (ii) it has to hold that $\text{len}(P', \tau) \geq \text{len}(P, \tau)$.

If G is time-independent, τ is ignored and the length of a path P is obtained by $\text{len}\, P$.

There are a few closely related problems which are variations of the Shortest Path Problem. We briefly mention the versions that are relevant for our work.

- Many-To-Many-Shortest Path Problem.
  This is a generalization of the Shortest Path Problem. Instead of one node s and t we are given a set of source nodes $S \subseteq V$ and a set of target nodes $T \subseteq V$. We now ask for a shortest path $P_{s,v}$ for each pair $(s,t) \in S \times T$. In multi-modal routing the Earliest Arrival Problem will actually transform to this version of the problem.

- One-To-All-Shortest Path Problem.
  This is a special case of the Many-To-Many-Shortest Path Problem where S is a singleton set consisting of one source node s and $T = V$ is the set of all nodes. Hence, we are asking for shortest paths P_v to every node $v \in V$. Because the edge set of all resulting paths $\mathcal{T} = \bigcup_{v \in V} P_v$ forms a tree, we might also say that we compute a *shortest path tree*.

- All-Pairs-Shortest Path Problem.
  This is a version of the Many-To-Many-Shortest Path Problem where both S and T are the complete node set V of the graph. Having the All-Pairs-Shortest Path Problem solved automatically includes solutions for all instances of the Shortest Path Problem in the graph. However, (pre-)computing shortest paths for all pairs of nodes tends to be very expensive both regarding memory consumption and execution time. Hence, this is not a viable approach.

Each of these versions of the Shortest Path Problem can be solved by Dijkstra's algorithm [Dij59]—possibly requiring repeated execution. Furthermore, as we work out in Sections 4.2.2 and 4.3.2, the algorithm can be augmented to work with time-dependent networks and even further to multi-modal networks.

We wind up this section by reducing the Earliest Arrival Problem to the Shortest Path Problem. We do this for each of our networks by applying the Shortest Path Problem to the graph obtained by the model we chose for the respective network (cf. Chapter 3). However, to keep things brief, we do not give extensive formal correctness proofs for the established models.

Road Networks. The road network is modeled in a straightforward way. A shortest s-t-path in the graph maps directly to a route between two intersections. Note that due to the two weights in our graphs (car and foot travel times), we need to decide beforehand which of the weights to evaluate. Using car weights for some and foot weights for other edges would correspond to mixing both modes of transportation.

Railway Networks. In the time-dependent railway network (with or without realistic transfer times), a (time-dependent) shortest path leads to an itinerary as follows. Each

time a connection edge is used in the network the interpolation point that was used for evaluating the edge yields a connection from the timetable that was used on the respective edge. Since the only way to get from one station to another is by using a connection edge (in each version of the model this is true), the sequence of obtained connections from the timetable forms a valid itinerary. Furthermore, if the path in the graph is a shortest path with some departure time τ from the source station, this itinerary is a solution to the Earliest Arrival Problem with departure time τ. More extensive proofs can be found in [BJ04] and [PSWZ07].

Flight Networks. Although the flight network is new, we can use the same correctness proof as for the realistic time-dependent railway model for large parts. Hence, we restrict ourselves to giving some informal notes.

Since two airports A_1 and A_2 are connected by a time-dependent flight edge in the graph if and only if there is a flight in the timetable from A_1 to A_2, a path in the flight graph always yields a valid itinerary. Like in railway graphs, the actual flight taken on that edge can be obtained from the interpolation point that was evaluated on the respective flight edge. Furthermore, the edges and nodes within the airport are exactly modeled to conform with the check-in, check-out and transfer time restrictions of each airport (depending on the flight model they are of miscellaneous complexity). Since these restrictions also apply to any itinerary wrt. the Earliest Arrival Problem, we eventually obtain that shortest paths in the flight graph correspond to itineraries solving the Earliest Arrival Problem and vice versa.

Mutli-Modal Networks. We have not made entirely clear how a best route on a multi-modal network looks like. If we allowed arbitrary (shortest) paths in the multi-modal network this could lead to undesirable results because not every mode of transportation may be available at any point in a real world scenario (for example a car between two trains). We will address this issue in Section 4.3.

4.2. Uni-Modal Routing

This section covers the classic case, where we have an uni-modal graph $G = (V, E)$. There have been extensive studies regarding routing in uni-modal networks, hence we only give a brief introduction of Dijkstra's algorithm and then turn toward multi-modal shortest paths.

Algorithm 3: DIJKSTRA's algorithm

Data: A weighted graph $G = (V, E)$, $s \in V$ and $T \subseteq V$.
Result: Shortest paths from s to all $t \in T$.

```
1  Q ← a priority queue of nodes
2  Q.insert(s, 0)
3  settled-targets ← ∅
4  while not Q.isEmpty() do
5      v ← Q.dequeue()
6      if v ∈ T then
7          settled-targets ← settled-targets ∪ {v}
8          if settled-targets = T then
9              stop ;                                  // shortest paths found
10     forall outgoing edges e = (v, w) do
11         if w is a new node then
12             Q.insert(w, dist_s(v) + w(e))
13             pre(w) ← v
14         else
15             if dist_s(v) + w(e) < dist_s(w) then
16                 Q.decreaseKey(w, dist_s(v) + w(e))
17                 pre(w) ← v
18 stop ;                                              // no shortest path found
```

s: a source node
T: taget nodes

e: edge
w: next node
w(e): edge weight
dist(v): distance from source node to node v
pre(w): preceding node of node w

4.2.1. Time-Independent Routing

Here we are given an uni-modal graph G with time-independent edge weights $w(e)$ on *all* edges $e \in E$. We present DIJKSTRA's algorithm first introduced in [Dij59]. See Algorithm 3 for pseudo code.

Terminology. We call a node $v \in V$ *settled* by the algorithm, if it has been extracted from the priority queue in Line 5. Note that a settled node never gets reinserted into the queue. Thus, each node is settled at most once. The set of all settled nodes during one run of the algorithm is called *search space*. An edge which is touched by the algorithm in Line 11 is called a *relaxed edge*. The label $\text{dist}_s(v)$ assigns each node its distance from the source s during computation and $\text{pre}(v)$ sets the preceding note

of v along the path. At the beginning, these two labels have to be initialized for all nodes $v \in V$ with $\text{dist}_s(v) = \infty$ and $\text{pre}(v) = \texttt{null}$, respectively. After the algorithm terminates, $\text{dist}_s(t)$ contains the length of the shortest s-t-path and the path can be constructed by walking along the pre-labels beginning at t until s is reached.

Solving Variants of the Shortest Path Problem. DIJKSTRA's algorithm solves each of the SHORTEST PATH PROBLEM variants introduced in Section 4.1.2. For single-source version of the problem, we stop as soon as *all* target nodes $t \in T$ have been settled (depending on T, this is either a single target, a set of targets or the whole graph). For the variant of the problem where we have a set S of source nodes, we repeatedly execute DIJKSTRA's algorithm for each of the source nodes $s \in S$ yielding shortest paths from s to all targets T per run.

However, regarding the multi-source multi-target version of the problem, we are often only interested in *one* path of minimal length from S to T. Hence, a single run of DIJKSTRA's algorithm is sufficient. Line 2 is substituted by inserting all nodes from S into the priority queue with key 0 and stop as soon as all target nodes from T have been settled. Selecting the path of minimal distance from all target nodes then yields the desired solution.

Negative Edge Weights. It shall be noted that for the correctness of DIJKSTRA's algorithm we requires the edge weights to be all positive or 0. Having negative edge weights may, in general, lead to cycles of negative length in the graph. There are other algorithms like the BELLMANN & FORD algorithm [Jr.56, Bel58] that can handle negative weights. However, in all of our graphs negative weights do not occur, hence, we do not look into this issue further.

4.2.2. Time-Dependent Routing

We now augment DIJKSTRA's algorithm to cope with time-dependency. We present two variants: Time queries to compute a shortest path for fixed departure time τ and profile queries, where we are interested in shortest paths for all times of day.

Time Queries. The time-dependent version of DIJKSTRA's algorithm is almost identical to the time-independent version as illustrated in Algorithm 3 when computing time queries. However, it is important that all edge functions on the input graph fulfill the FIFO-property (cf. Section 3.3.4, page 25). The only changes to the algorithm that need to be made are the following two.

- We need to supply a departure time τ as additional input.

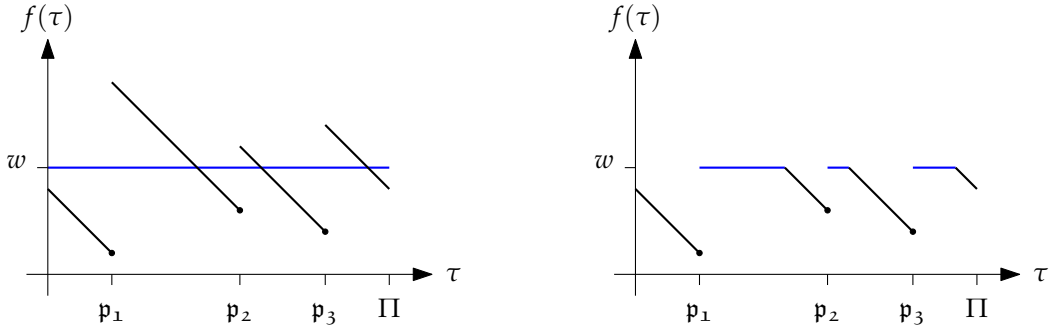
- To evaluate the edge weights, we have to consider the current time at which we encounter the respective edge. Let $e = (v, w)$ be an edge of which the weight has to be evaluated (cf. Lines 12, 15 and 16). Then the time at which we evaluate the function f_e of the edge e is the departure time τ plus the distance along the path to v (available to our algorithm by $\text{dist}_s(v)$). Hence $w(e)$ has to be replaced by $f_e(\tau + \text{dist}_s(v))$ at all occurrences in Algorithm 3.

Profile Queries. Using the previously described version of the time-dependent query algorithm yields only shortest paths for *one* particular departure time τ. While this seems to be a canonical generalization of the time-independent case, there is another type of query in time-dependent graphs, where we are not only interested in the shortest path at one time point, but at all times of day.

For example in a railway network we state 8 o'clock as departure time τ for a query. Let's say there is a train departing at 8:00 to our destination takes 2 hours. But maybe there is another train departing at 9 o'clock that takes only 1 hour and 10 minutes. Taking the second train would be a suboptimal solution to the Earliest Arrival Problem (since the arrival time is 10 minutes later), but its sheer travel time is 50 minutes shorter. So, maybe it would be nice to present the user with the travel time for each possible departure time $\tau < \Pi$. In other words, the result of the query should be a piecewise linear function f itself, where each interpolation point represents a shortest path for that particular time. Besides generating user friendly output, profile queries are necessary for some of our speed-up techniques where the distance of a node at all times of day is required.

Augmenting Dijkstra's Algorithm. Algorithms for solving profile queries in time-dependent networks have been introduced in [Dea99]. According to [Dea99], a profile s-t-query can be computed as follows. Let $\text{dist}_s^*(v)$ denote the profile function containing the distance of v from s at all times of day. Then we modify Dijkstra's algorithm by using $\text{dist}_s^*(v)$ as distance labels on the nodes. As key of the priority queue we use the *lower bound* $\underline{\text{dist}}_s^*(v)$ of the respective distance label f. In the beginning, the source node s is inserted with the (constant) zero-function f_0 as label, while all other nodes are initialized with f_∞. Then for each extracted node v with distance label $f_v := \text{dist}_s^*(v)$ we iterate over all outgoing edges $e = (v, w)$ and compute the temporary function $f_w^{\text{new}} := f_v \oplus f_e$ (link-operation) where f_e denotes the function assigned to the edge e.

Now, this is the crucial difference to the classic variant of Dijkstra's algorithm: If $f_w^{\text{new}} \geq f_w$ does *not* hold, then f_w^{new} is an improvement at least for one departure time over the day. Hence, we insert w into the priority queue with the label $\min(f_w, f_w^{\text{new}})$ (merge-operation). There are two things to be noted: First, an untouched node is



(a) Before the merge. **(b)** After the merge.

Figure 4.1.: This figure illustrates the problem when merging a piecewise linear function (black) with a time-independent weight w (blue). The left side shows both functions in their original form, while the right side shows the resulting function after computing $\min(f, w)$. The outcome is no longer homogeneous, as not all segments of the functions have the same gradient γ.

always inserted since $f_w^{\text{new}} \geq f_w$ never holds for $f_w = \infty$. Second, it is possible that a node is inserted into the queue multiple times during one run of the algorithm. Thus, the stop criterion must be modified. We can only stop as soon as the lowest key in the priority queue is larger than the upper bound of the function f_t assigned to the target node t. Then we can guarantee that there is no undiscovered, yet improving path to t.

Problems. The above algorithm makes use of both the link- and merge-operations. A problem arises when our graphs contain time-dependent edges using piecewise linear functions with different gradients. While the time-dependent edges modeling rail or flight connections have a gradient of -1, our graphs also contain 'constant' edges that are in fact represented as constant functions having gradient 0 (e.g., in the railway model the edges from station to route nodes). While linking edges of different types is unproblematic, merging them yields an inhomogeneous piecewise linear function, where not all segments are of the same gradient. See Figure 4.1 for an illustration on this matter.

Solution. Fortunately, the merge-operation is not compulsory. Algorithm 4 illustrates a modification of the previously described profile search algorithm. Algorithm 4 is called a *multi label correcting* algorithm. Instead of one label per node, we assign each node a set of labels. These labels are assigned to the queue items (When `dequeue` is called, the queue item is popped off but not deleted, hence, the labels are still available afterward. See also Appendix A.4). For each settled node we, thus, compare

Algorithm 4: Multi Label Correcting Algorithm

Data: A weighted time-dependent graph $G = (V, E)$, $s \in V$ and $T \subseteq V$.
Result: Profiles from s to all $t \in T$.

```
1  Q ← a priority queue of nodes
2  Q.insert((s, f_0), 0);            // Insert source with 0-function as label
3  while not Q.isEmpty() or not isFinished() do
4      (v, f) ← Q.dequeue()
5      forall outgoing edges e = (v, w) do
6          f_w^new ← f ⊕ f_e;                // compute temporary new label at w
7          ins ← true
8          forall labels f_w on w do
9              if f_w ≤ f_w^new then
10                 ins ← false
11             else if f_w ≥ f_w^new then
12                 removeLabel(w, f_w)
13         if ins is true then
14             Q.insert((w, f_w^new), f_w^new);    // Use lower bound of f_w^new as key
15 stop
```

the temporary function f_w^{new} to *all* labels assigned to the target node w (cf. Line 8) and (re-)insert the node if there is no label assigned to w that dominates f_w^{new} completely (cf. Line 9). On the other hand, if the new label f_w^{new} dominates an existing label f_w completely, f_w can be deleted (cf. Line 11). The stop criterion (we omit it in the figure for clarity) has to be modified, as we now need to use the maximum of all upper bounds assigned to the target nodes t. Likewise, the profile function at target t can be computed by merging all labels of the respective target node.

Outputting Paths. It shall be noted that the output of each of the algorithms is a profile function f_t that yields the travel time $f_t(\tau)$ from s to t for all $\tau < \Pi$. We did not mention how the actual shortest path can be obtained for a certain time point τ. The first option is to use the time-dependent version of Algorithm 3 on τ. Alternatively, path information can be integrated into the link-operation of two functions. When two functions are linked, we assign the path information to the interpolation points, which can then be used to unpack the path for a given interpolation point $\mathtt{p}$. We omit going

(a) Bad path. **(b)** Good path.

Figure 4.2.: Comparison between two routes. Black edges are car-edges, red edges railway-edges and blue edges foot-edges. The bold path to the left is the shortest path from s to t. While this path may be the fastest, it is undesirable since we are required to use a car in the middle of our journey. So the path to the right should be chosen.

into further details since for those cases where we have to compute profile queries in this work, the path information is irrelevant.

Running Time. Regarding time-complexity, we anticipate an increase compared to time queries since nodes are inserted multiple times. Additionally, the link-operation is expensive and the stop criterion (which has to be checked each iteration) reasonably more complex. In [Dea99] it has been shown that the number of (re-)insertions heavily depends on the nature of the edge-functions. We observed ourselves that for rather small networks which have only a few thousand nodes, the number of `dequeue` operations is in the magnitude of millions per s-t profile query. Together with the increased time complexity when using operations on functions instead of scalars, this makes our multi label correcting algorithm only feasible for very small networks.

4.3. Multi-Modal Routing

In the previous section we developed the basic ingredients for solving the Shortest Path Problem on both time-independent and time-dependent networks. In this section we proceed by adapting these techniques to the multi-modal case. However, it turns out that 'classic' shortest paths may correspond to very undesirable routes. Hence, we augment the Shortest Path Problem by introducing constraints on the sequence of labels that shortest paths must comply to and model these constraints by (regular) languages.

In all subsequent sections of this chapter we are given a multi-modal graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ having mixed time-dependent and time-independent edge weights. Furthermore, the label function assigns each node and edge its type (for example `ROAD_NODE` or `RAIL_EDGE`). The sets of node and edge labels are denoted by $\mathcal{L}_{\text{node}}$ and $\mathcal{L}_{\text{edge}}$, respectively.

Undesirable Paths. Solving the Shortest Path Problem yields nice routes (resp. itineraries) on uni-modal graphs. However, when these graphs are combined, a shortest path can lead to an unwanted route. Imagine the following. Let's assume, we have a multi-modal graph consisting of a road and a railway network. If we are planning a far journey and do not want to use the car all the way, we probably think of going to the nearest railway station (by car or foot), then take a train to the destination city and take the last part of the journey by foot or taxi. So what just happened here, is that we had some rather strict opinion in mind as to how a 'good' path should look like. If we computed our request by a uni-modal shortest path algorithm, the outcome could be very different, however. For example in the middle of the railway line there is a new highway which runs parallel to the tracks. So, regarding pure travel time, it would be beneficial to get off the train, use the new highway (by car) and re-board another train at the end of the highway. In general, this can happen arbitrarily often. Such a path, while being the fastest, is useless since we probably do not have a car available in the middle of the journey.

Figure 4.2 illustrates the issue with an example where we have the transportation modes *car*, *railway* and *foot*. In general, arbitrary modes of transportation at arbitrary points of the network should be avoided.

4.3.1. The Label Constrained Shortest Path Problem

The Label Constrained Shortest Path Problem has been studied in [BJM00] and is an augmentation of the classic Shortest Path Problem. Before explaining how it can be used for multi-modal routing, we give its abstract definition and cite some results regarding the theoretical complexity of the problem.

Definition 5 (Label Constrained Shortest Path Problem)**.** *Given an alphabet Σ, a language $L \subset \Sigma^*$, a weighted, directed graph $G = (V, E)$ with Σ-labeled edges and source and target nodes $s, t \in \Sigma$, we ask for a shortest path P from s to t, where the sequence of labels along the edges of the path forms a word of L. Thus given $P = [v_1, \ldots, v_k]$ it has to hold that*

$$\text{label}\big((v_1, v_2)\big)\,\text{label}\big((v_2, v_3)\big)\cdots\text{label}\big((v_{k-1}, v_k)\big) \in L. \tag{4.1}$$

The general problem formulation demands no restriction on the language L. An extensive theoretical study has been conducted on the complexity of the problem regarding different types of languages [BJM00]. For our application of the problem, however, regular languages (cf. Section 2.2) are sufficient to model reasonable path restrictions. In [BJM00] the following theorem has been proven.

Theorem 1. *The* RegL-CSPP *(*Label Constrained Shortest Path Problem *restricted to regular languages) can be solved in deterministic polynomial time.*

This lays the basic foundation for an efficient algorithm, which is introduced in the next section.

4.3.2. Algorithms

A polynomial time algorithm for the RegL-CSPP has been first introduced in [BJMoo]. It operates on a product network of the input graph $\boldsymbol{G}$ and a (non-deterministic) finite automaton describing the language L. In [Holo8] and [BBH$^+$o8] this algorithm has been optimized, so the product network does not need to be computed explicitly. This reduces the space consumption significantly. We adapt this algorithm to our time-dependent multi-modal scenario in this section. Furthermore, from this point on we identify the set of edge labels $\mathcal{L}_{\text{edge}}$ with Σ.

Product Network. Since our algorithm operates on a product network of $\boldsymbol{G}$ and a finite automaton describing the language constraint, we give a formal definition of product networks between these two structures.

Definition 6 (Product Network)**.** *Given a Σ-labeled graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ and a non-deterministic finite automaton $\mathcal{A} = (Q, \Sigma, \delta, S, F)$, the* product network $G^\times = (V^\times, E^\times)$ *is defined as follows.*

- *The node set consists of* product-nodes $(v, q) \in V^\times$ *where $v \in \boldsymbol{V}$ and $q \in Q$.*
- *An edge $e^\times = \big((v_1, q_1), (v_2, q_2)\big)$ between two product-nodes is included in $E^\times$ if and only if $e = (v_1, v_2) \in \boldsymbol{E}$ and there is a label $\sigma \in \Sigma$ for which exists a transition $q_2 \in \delta(q_1, \sigma)$ in the automaton. The weight of $e^\times$ is set to the weight of e and* $\text{label}(e^\times)$ *is set to σ.*

The resulting graph $G^\times$ is uni-modal.

In [BJMoo] the following theorem has been proven which automatically leads to an algorithm.

Theorem 2. *The* RegL-CSPP *for a Σ-labeled graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ from source $s \in \boldsymbol{V}$ to target $t \in \boldsymbol{E}$ and a regular language $L \subseteq \Sigma^*$ can be reduced to the* Shortest Path Problem *as follows.*

1. *Construct a (non-deterministic) finite automaton $\mathcal{A} = (Q, \Sigma, \delta, S, F)$ describing L.*
2. *Construct the product network $G^\times = \boldsymbol{G} \times \mathcal{A}$.*

3. *Solve the* Many-To-Many-Shortest Path Problem *on* $G^{\times}$ *with source node and target node sets*

$$S := \bigcup_{q_s \in S} (s, q_s) \quad \textit{and} \quad T := \bigcup_{q_f \in F} (s, q_f). \tag{4.2}$$

4. *From all resulting paths pick the one having minimal length.*

Let $P = [(v_1, q_1), \ldots, (v_k, q_k)]$ be the shortest path obtained by the algorithm induced from Theorem 2. Then the length of the path in $\boldsymbol{G}$ is the same as the length $\text{len}(P)$ in $G^{\times}$. The actual path in $\boldsymbol{G}$ can be obtained by omitting the 'automaton part' of the product-nodes, thus, yielding $[v_1, \ldots, v_k]$. On the other hand, the word conforming to L along the path can be obtained by concatenating the edge labels

$$\text{word}(P) := \text{label}\big((v_1, q_1), (v_1, q_2)\big) \cdots \text{label}\big((v_{k-1}, q_{k-1}), (v_k, q_k)\big). \tag{4.3}$$

Complexity. Step 1 can be computed in polynomial time. The product network in step 2 can be computed in time $\mathcal{O}(|\boldsymbol{G}| \cdot |\mathcal{A}|)$ which is also polynomial. Hence, the algorithm induced by Theorem 2 runs in polynomial time. However, regarding memory complexity, the space required to store the product graph $G^{\times}$ is also in $\mathcal{O}(|\boldsymbol{G}| \cdot |\mathcal{A}|)$ which is too much to fit into memory for our largest instances of our multi-modal networks.

Implicit Computation of the Product Network. Fortunately, the algorithm from Theorem 2 can be augmented in such a way that $G^{\times}$ does not need to be computed explicitly in advance. Instead, we use $\boldsymbol{G}$ and the transition graph of $\mathcal{A}$ separately as input, thus, reducing the input space complexity to $\mathcal{O}(|\boldsymbol{G}| + |\mathcal{A}|)$. The product network is then computed *implicitly* only for the nodes and edges that are touched by Dijkstra's algorithm. In the worst case Dijkstra's algorithm still visits the whole graph, thus, not improving on the theoretical complexity bound. However, these cases are rare and were not observed during our experiments.

Algorithm 5 shows the final method. For the sake of simplicity, we illustrate the algorithm using solely time-independent weights (Note, our multi-modal graphs are in fact partially time-dependent). Furthermore, we omitted the assignments of $\text{pre}(\cdot, \cdot)$ and $\text{dist}_s(\cdot, \cdot)$ in the figure. Note that in contrast to the previous illustrations of shortest path algorithms, we denote the priority queue with PQ in order to avoid confusion with the set Q of states of the finite automaton.

The algorithm matches Algorithm 3 except for the following differences. The priority queue contains product-nodes instead of regular nodes and is initialized with all product-nodes composed of the source node s and any of the initial states of the finite

Algorithm 5: Time-Independent Multi-Modal Dijkstra

Data: A multi-modal graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$, $s \in \boldsymbol{V}$ and $T \subseteq \boldsymbol{V}$. Further, a finite automaton $\mathcal{A} = (Q, \Sigma, \delta, S, F)$ representing a regular language $L \subset \Sigma^*$.

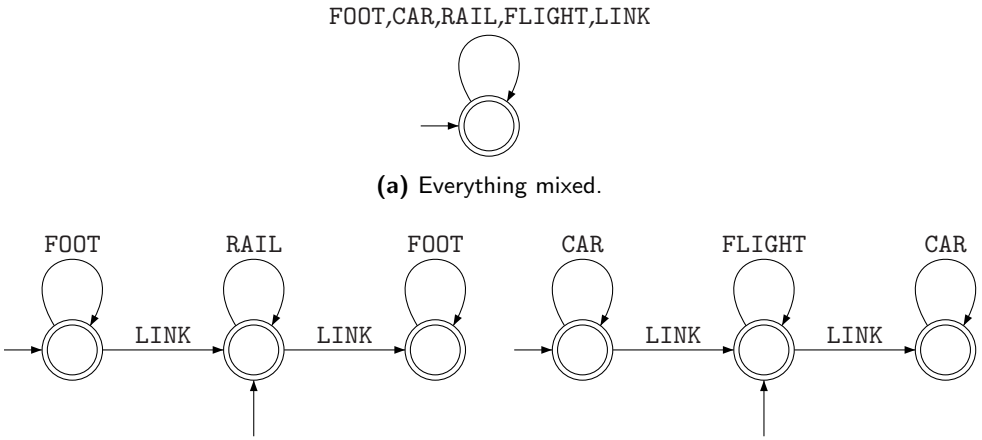
Result: A shortest path from s to t wrt. to L.

```
1  PQ ← a priority queue of product-nodes
2  forall q_s ∈ S do
3      PQ.insert((s, q_s), 0)
4  settled-targets ← ∅
5  while not PQ.isEmpty() do
6      (v, q) ← PQ.dequeue()
7      if v ∈ T and q ∈ F then
8          settled-targets ← settled-targets ∪ {v}
9          if settled-targets = T then
10             stop ;                                  // all shortest paths found
11     forall outgoing edges e = (v, w) do
12         forall states q' ∈ δ(q, label(e)) do
13             if (w, q') is a new product-node then
14                 PQ.insert((w, q'), dist_s((v, q)) + w(e))
15             else
16                 if dist_s((v, q)) + w(e) < dist_s((w, q')) then
17                     PQ.decreaseKey((w, q'), dist_s((v, q)) + w(e))
18 stop ;                                              // not all shortest paths found
```

automaton (cf. Line 2). Accordingly, in Line 7 it is not sufficient to settle a target node. Moreover, the automaton has to be in a final state.

The 'implicit computation' of the product graph $G^\times$ occurs in Lines 11 and 12: For the currently considered product-node (v, q), we simultaneously iterate over both the outgoing edges of v in $\boldsymbol{G}$ and the outgoing transitions in $\mathcal{A}$ from state q labeled by the edge label label(e), hence 'simulating' walking along the edge $e^\times = \big((v, q), (w,' q)\big)$ of the product network. Note that the 'virtual' edge $e^\times$ would also exist in $G^\times$ and that every edge that exists in $G^\times$ is also considered by this method, thus, the algorithm processes exactly the edges of $G^\times$.

Regarding the stop criterion in Line 7, we obtain a shortest path to t, as soon as



(a) Everything mixed.

(b) Foot & Railways.

(c) Car & Flights.

Figure 4.3.: Several automata.

the *first* product-node (t, q_f) has been settled (There could exist further final product-nodes (t, q'_f) with $q_f \neq q'_f$). This is due to the fact that according to step 4 of Theorem 2, we only require the shortest path of minimal length between all pairs of source and target nodes in the product network $G^\times$.

Complexity. Regarding time complexity we anticipate an increase of a factor of approximately $|\mathcal{A}|$ when compared to the uni-modal version of Dijkstra's algorithm on G. This is due to the fact that the search space increases by approx. $|\mathcal{A}|$ as there are approx. $|\mathcal{A}|$ times more nodes in the (implicit) product network. Furthermore, the whole search space needs to be kept in memory in order to have dist and pre available for all product-nodes.

Finite Automata and Examples. We kept the description of both the problem formulation and the algorithms rather brief. For example, we did not restrict ourselves to a specific automaton or a class of automata. This is intentional. However, because of their relevance, there are a few interesting automata we would like to point out. Have a look at the Figures 4.3 and 4.4.

Accepting Everything. The automaton in Figure 4.3a is very simple. It has only one state and accepts Σ^*. Using this automaton with Algorithm 5 yields virtually a classic Dijkstra run. This can be used to examine the performance loss of the multi-modal Dijkstra variant compared to the classic version due to the more complex data structures (see Section 6.3 on this).

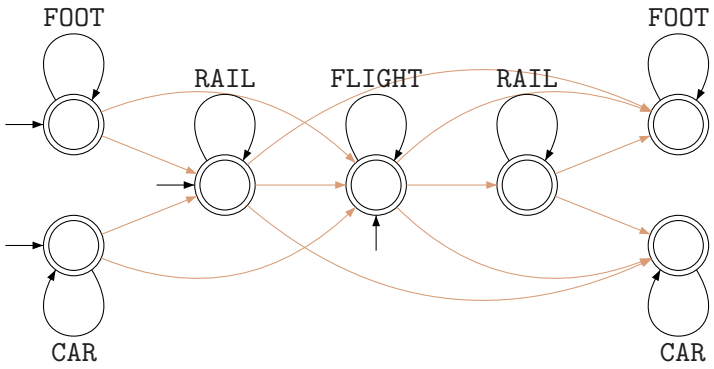


Figure 4.4.: Reasonably incorporating every mode of transportation in a hierarchical approach. For clarity, LINK_EDGEs are not labeled, but colored in light red.

Foot and Rail. This automaton shown in Figure 4.3b accepts the language $f^*lr^*lf^*$, where f is short for FOOT_EDGE, l is short for LINK_EDGE and r is short for RAIL_EDGE. This models a common scenario when we plan a journey of medium length (for example from south to north Germany) and we do not have a car available.

Car and Flight. See also Figure 4.3c. This automaton is almost equivalent to the foot and rail automaton, except we use car and flight, instead. The accepted language is $c^*lg^*lc^*$ (c being CAR_EDGE and g being FLIGHT_EDGE).

Hierarchical Automata. While in principle we could use any arbitrary language L for our multi-modal queries, not every language makes sense. For example, allowing only a finite number of (non-link) edges at some point is absurd, as this would mean demanding to use a fixed predefined number of edges. So, a 'meaningful' automaton probably consists of states having a loop labeled by one of the edge labels $l \in \Sigma \setminus \{\texttt{LINK\_EDGE}\}$. Since we linked the different networks together by LINK_EDGEs (cf. Section 3.5.2), the states should be connected by a LINK_EDGE labeled transition. Meanwhile, the arrangement of the particular states, can be done in many reasonable ways.

If we take a look at the different network types, they contain an inherent hierarchy, when ordered as follows:

$$\text{Road network} \prec \text{railway network} \prec \text{flight network}. \tag{4.4}$$

When planning a typical multi-modal voyage we observe that in most cases we probably start at the lowest point in the hierarchy (by car or foot), step up by switching modes of transportation several times and finally step down in the hierarchy at the end of the journey. This is characterized by the following definition on languages.

Definition 7. *Let $L \subset \Sigma^*$ be a language. Then L is called a* hierarchical language *if it holds that for every word $w \in L$ containing two symbols $\sigma_i, \sigma_j \in w$ with $i > j$ where $\sigma_j \prec \sigma_i$ wrt. the hierarchy (4.4), there must not be a third symbol $\sigma_k \in w$ with $k > j$ and $\sigma_k \succ \sigma_j$. An automaton which accepts a hierarchical language is called a* hierarchical automaton.

In short, after once descending in the hierarchy we may not ascend again. An automaton that covers this aspect is presented in Figure 4.4. Note that we allow either foot or car as the first and last mode of transportation. Furthermore, we allow skipping modes in the hierarchy as well as using any mode of transportation in the beginning and the end. Hence, we also allow routes that only involve a sub-hierarchy.

4.4. Summary

In this section we presented basic routing algorithms for both uni-modal and multi-modal networks. We did this by starting with the classic time-independent uni-modal scenario, augmenting the problem via uni-modal time-dependent routing to multi-modal routing. These are the main results.

- We use the Earliest Arrival Problem to model (timetable information) queries on all networks. For some source s, target t and departure time τ we are interested in a route from s to t that arrives as early as possible. This problem is equivalent to the (time-dependent) Shortest Path Problem on each of the networks.

- The uni-modal time-independent Shortest Path Problem can be solved by Dijkstra's algorithm. Variants of the Shortest Path Problem, where we have multiple source or target nodes, can be solved by (possibly multiple runs of) Dijkstra's algorithm as well.

- In the time-dependent case, there are two interesting types of queries. Time queries where we ask for a shortest s-t-path for a specific departure time τ and profile queries where we are interested in a travel time function that yields the travel time from s to t for any given time of day. While the first can be solved by a straight-forward augmentation of Dijkstra's algorithm, the latter requires a label correcting algorithm that propagates piecewise linear functions through the network. Furthermore, nodes may be inserted into the priority queue multiple times, which results in a significantly worse time complexity of the algorithm.

  Because the merge-operation on functions between time-independent (constant) and time-dependent functions is not supported in our implementation, we pre-

sented a multi label correcting algorithm which manages without the need for the merge-operation at the price of storing multiple labels on each node.

- Applying an uni-modal routing algorithm on a multi-modal network can result in unrealistic paths (cf. Figure 4.2). For that reason, we introduced the Label Constrained Shortest Path Problem where the edges in the network are labeled according to their transportation mode and the concatenated labels of a shortest s-t-path must constitute a word to some predefined language L over Σ.

  When only allowing regular languages for L, the problem is solvable in polynomial time by using Dijkstra's algorithm on the product network of the multi-modal input graph G and the transition graph of the finite automaton accepting L. The (rather theoretical) algorithm on the product network can be refined to compute the product-nodes implicitly. This saves a lot of space which makes the problem manageable for large networks regarding memory consumption.

- The running time of the multi-modal routing algorithm depends on the complexity on the finite automaton (see also Section 6.3). Furthermore, we introduced hierarchical languages which resemble a common way to combine transportation modes.

Chapter 5

Speed-Up Techniques

In the previous chapter we introduced basic routing algorithms to solve the Earliest Arrival Problem on our models. These algorithms are all based on Dijkstra's algorithm which was already introduced in 1959 [Dij59]. However, only over the last years computer hardware evolved far enough to allow the handling of large scale networks like those encountered in route planning. Through experiments it turns out that even on todays hardware query times are too high. For example, a random query between two nodes in the German road network takes several seconds on contemporary server hardware. This fact gets even worse when we think of route planning on mobile devices such as sat-nav systems for cars.

To encounter this problem, research in the past years focused on developing *speed-up techniques* (mostly optimized on road networks) for Dijkstra's algorithm which all have the same goal of reducing the search space while still yielding provably optimal results. A brief overview over recent development is given in [DSSW09a]. It turns out that there are a few basic ingredients on which most speed-up techniques are based [Del09a]. These can be abstracted by bi-directional search, goal-directed search and contraction.

In this chapter we first present each of the basic ingredients: Bi-directional search, contraction and goal-directed search by introducing their algorithmic concepts in the context of uni-modal time-independent routing. Moreover, we either describe how these concepts can be adapted to multi-modal routing, or present why this is difficult.

While bi-directional routing only requires a modification to the query algorithm, both contraction and goal-directed search are requiring a two-phased process. First a preprocessing routine computes additional data on the input. Second, this data is then used by the query algorithm to reduce the search space. Regarding goal-directed search, we introduce two concepts: A* with landmarks (ALT) and Arc-Flags. Both

techniques are goal-directed in the sense that they narrow the search space toward the goal (the target t), but they are using a quite different method, from which ALT turns out to be easily adaptable to multi-modal routing, whereas Arc-Flags is not.

As it turns out that there is a trade-off between developing a very fast speed-up technique and the general applicability of the technique regarding arbitrary automata, we focus on both sides of the coin. First, we introduce multi-modal *Core-Based Routing*, which is a method belonging to the category of contraction techniques. Basically, we compute a *core graph* through bypassing nodes and inserting shortcuts. The size of the core graph is only a small portion of the original graph. The query algorithm briefly works as follows. From the source and target we search for entry points into the core. Those are found very fast. The main search is then performed on the much smaller core graph by applying a standard Dijkstra algorithm from the entry- to the exit-points of the core.

The advantage of Core-Based Routing is the possibility to combine it easily with a goal-directed approach. Instead of using simple Dijkstra on the core, we replace it by a goal-directed technique. As Arc-Flags turns out difficult to adapt to multi-modal networks, we combine Core-Based Routing with ALT (Core-ALT). Core-ALT. Note that Core-ALT is not new and has been introduced in [BDS+08], however, we present a multi-modal variant of Core-ALT.

Finally, we drop our demand of applying arbitrary automata on the query algorithm, but instead focus on a reasonable subclass of automata where the road network is only used at the beginning and the end of the journey. We present a new multi-modal speed-up technique called *Access-Node Routing* which adapts some of the ideas behind Transit-Node Routing [BFM+07, BFSS07]. As it turns out that most of the time spent during a multi-modal query is in the large road network, the basic idea is the following. For each node v in the road network, we precompute a set $A(v)$ of *access-nodes* that are basically entry points into the public transportation network. Hence, the query algorithm is working in two phases. First, for the source s and target t (in the road network) we obtain their access-nodes by performing a table look-up. In the second part of the query we then perform a classic Dijkstra-search from the access-nodes $A(s)$ of s to the (backward) access-nodes $\overleftarrow{A}(t)$ of t in the public transportation network. Th partial paths are combined in the end. Regarding local queries that do not use the public transportation network, we fall back to computing a time-independent uni-modal shortest path search restricted to the road network.

We like to mention, that the main contribution from Access-Node Routing is not the speed-up gained by skipping the road network during the query (although this fact alone yields high speed-ups enabling us to compute intercontinental queries on large scale networks in a few milliseconds; also see Section 6.5). Rather, through

Access-Node Routing we are able to isolate the public transportation network from the road network in the sense that separate algorithms can be applied on the public transportation and the road networks. For local queries we can use one of today's high-performance speed-up techniques which are able to compute local queries in microseconds time. On the public transportation network, on the other hand, we could use any other time-dependent (multi-modal) speed-up technique or even another algorithm like for example multi-criteria search.

Access-Node Routing is explained in the last section of this chapter.

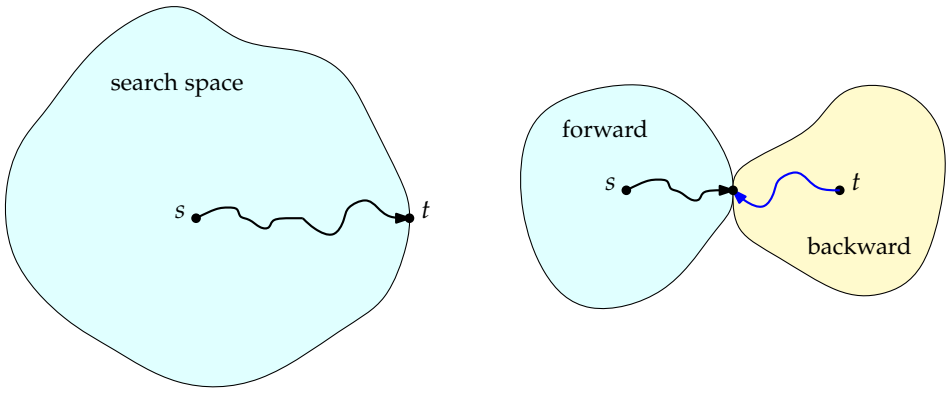
5.1. Basic Ingredients

We start off with listing the basic ingredients bi-directional routing, goal-directed search and contraction since these concepts underly our speed-up techniques. We thereby explain the preprocessing and the query algorithm separately. For simplicity, we describe the ingredients on the basis of time-independent networks and present the augmentation to time-dependency afterward. Finally, we discuss for each ingredient how it can be adapted to multi-modal queries or explain why this is difficult.

5.1.1. Bi-Directional Search

When computing a shortest s-t-path using Dijkstra's algorithm (cf. Algorithm 3), the search starts at s and we subsequently settle nodes v until t is settled. Note that if a node v is settled one time, it is never settled again, hence $\text{dist}_s(v)$ is the shortest distance to v (from s) and it holds for all nodes u that were settled before v that $\text{dist}_s(u) \leq \text{dist}_s(v)$. By these means, the search can be imagined as growing a ball around s during the execution of the algorithm.

The idea behind bi-directional routing is that besides executing an s-t-query in G, we also perform a query from t to s in the backward graph $\overleftarrow{G}$ (see also [Dan62, GH05]). The s-t-search in G is called *forward search* and the t-s-search in $\overleftarrow{G}$ *backward search*. The algorithm terminates as soon as either the forward or the backward search settles a node that has already been settled by the particular other search. Let v be the node where both search spaces meet. Then the shortest path P from s to t in G is the composition of the (implicitly computed) shortest path from s to v by the forward search and the shortest t-v-path computed by the backward search. Note that the edges of the t-v-path in $\overleftarrow{G}$ have to be flipped in order to obtain the correct path. Figure 5.1 visualizes the comparison between the search space sizes of the uni-directional and bi-directional algorithms.



(a) Uni-directional Dijkstra. **(b)** Bi-directional Dijkstra.

Figure 5.1.: Comparing uni-directional with bi-directional search. The search space grows like a 'ball' around s (resp. t). When searching from s and t simultaneously the search space can be reduced, as the algorithm may stop as soon as a node is in the search space of both the forward and backward searches.

Query. The query algorithm is basically a meta-algorithm composed of two DIJKSTRA algorithms: One for the forward s-t-search and one for the backward t-s-search. Each algorithm has to flag the nodes belonging to its respective search space and the other algorithm needs to have access to it. The meta-algorithm then controls the forward and backward search by alternately running one iteration in each algorithm. The algorithms terminate as soon as the search spaces meet. This yields a reduction in search space as the sum of the individual search spaces of the forward and backward search are generally smaller than the search space of a single forward search [BDW07].

Time-Dependency. Adapting bi-directional search to time-dependent networks with regard to time queries turns out difficult. This is due to the fact that the *arrival time* at the target node t is not known in advance. Hence, we have no basis for starting a backward search in $\overleftarrow{G}$. Still, there is a chance where the backward search may at least 'assist' the forward search. The forward search is time-dependent with departure time τ, while the backward search from t is performed on the time-independent lower bound graph $\underline{\overleftarrow{G}}$ from t.

Let $\mathcal{S}$ denote the set of nodes inside the search space of the forward search and $\overleftarrow{\mathcal{S}}$ the set of nodes discovered by the backward search. Then as soon as the two search spaces meet, i.e., $\mathcal{S} \cap \overleftarrow{\mathcal{S}} \neq \emptyset$, we compute a preliminary time-dependent shortest path to t. Let v be the node where the two search spaces met. The preliminary s-t-path P_{prel} is obtained by evaluating the v-t-path induced by the backward search

time-dependently with respect to the arrival time at v obtained by the forward search. Note that $\operatorname{len} P_{\text{prel}}$ is only an *upper bound* for the actual distance from s to t.

In the next step, the time-dependent forward and the time-independent backward search continue. The backward search may only be stopped if there is a node $v \in \overleftarrow{\mathcal{S}}$ with $\overleftarrow{\operatorname{dist}}_t(v) \geq \operatorname{len} P_{\text{prel}}$. In this case it can be guaranteed, that the shortest s-t-path is contained in $\mathcal{S} \cup \overleftarrow{\mathcal{S}}$. Finally, solely the forward search continues until t is settled, however, restricted to the node induced subgraph of $\overleftarrow{\mathcal{S}}$.

We conclude that time-dependent bi-directional search is much more complicated. In fact, using time-dependent bi-directional search may lead to speed-downs when compared to the simple time-dependent uni-directional Dijkstra algorithm—even when combined with goal-directed search (ALT) [NDLS08].

Adaption to Multi-Modal Routing. Adapting bi-directional search to multi-modal routing is straightforward. The forward and backward searches are instances of the multi-modal query algorithm as introduced in Section 4.3.2 (cf. Algorithm 5). Let $\boldsymbol{G}$ be a multi-modal graph, $\mathcal{A}$ a finite automaton and $S, T \subseteq \boldsymbol{V} \times Q$ be the sets of source, resp. target product-nodes wrt. to Theorem 2 on page 53. Like in the uni-modal case, the forward search performs a simple S-T-search while the backward search performs a T-S-search. The algorithm may then stop as soon as a product-node (v, q) is settled by both searches.

Note that in order to obtain valid paths that conform to the regular language represented by $\mathcal{A}$, the backward search not only needs to run on the backward graph $\overleftarrow{\boldsymbol{G}}$, but also on the *inverse automaton* $\overleftarrow{\mathcal{A}}$. Given a finite automaton $\mathcal{A} = (\Sigma, Q, \delta, S, F)$, the inverse automaton $\overleftarrow{\mathcal{A}} := (\Sigma, Q, \overleftarrow{\delta}, \overleftarrow{S}, \overleftarrow{F})$ is defined as follows. The start and final states are flipped, thus, $\overleftarrow{S} := F$ and $\overleftarrow{F} := S$ and the transition function is inverted by the law

$$q' \in \overleftarrow{\delta}(q, \sigma) \quad \Leftrightarrow \quad q \in \delta(q', \sigma). \tag{5.1}$$

If the multi-modal graph $\boldsymbol{G}$ is completely time-independent, this yields correct solutions to the multi-modal Earliest Arrival Problem with respect to the language L. However, in our scenario $\boldsymbol{G}$ is both time-independent and also time-dependent. In this case the same restrictions as to the uni-modal case apply: The arrival time is not known in advance, hence, we can only use the backward search to 'assist' the forward search by the means described in the previous paragraph.

Discussion. Bi-directional routing is one of the most basic ingredient, as many speed-up techniques make use of it. In our work we use a modified bi-directional search for the Core-Based Routing approach (cf. Section 5.2). However, we use this technique

only on time-independent parts of the multi-modal network (only in the road network, to be specific). Hence, the problems occurring when time-dependency has to be accounted for, do not apply in our case.

5.1.2. A* with Landmarks (ALT)

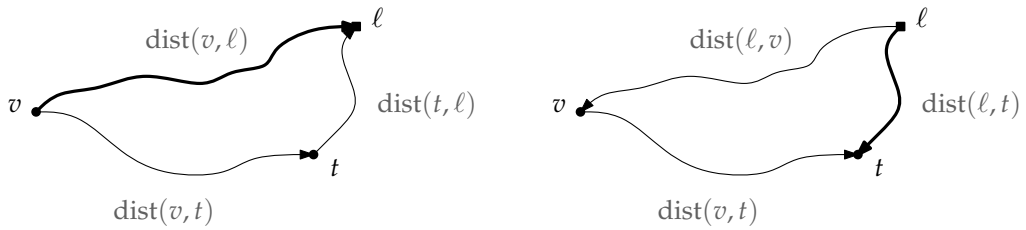
In this section we describe the first goal-directed search algorithm. The method is a combination between **A***-search, introduced in [HNR68] with the improvement of using **L**andmarks and the **T**riangle inequality [GH05, GW05].

A* is goal-directed in the sense that it preferably settles nodes that are closer to the target. This is done by altering the priority of the nodes according to a potential function which has the effect of modifying the order in which they are settled by DIJKSTRA's algorithm, hence, 'pushing' the search toward the target.

Potential Functions. Consider $\pi : V \to \mathbb{R}$ to be a *potential function* from the node set of the graph into the reals. We alter the weights of G to *reduced costs* by applying π to the edge weights by $w_\pi(u,v) := w(u,v) + \pi(v) - \pi(u)$. The graph with all edges altered is denoted by G_π. Then the length of any path (including shortest paths) $P = [v_1, \ldots, v_k]$ in G changes to $\text{len}_\pi(P) = \text{len}(P) + \pi(v_k) - \pi(v_1)$, because on subsequent edges the potentials cancel out each other. DIJKSTRA's algorithm only works on non-negative edge weights, so arbitrary potential functions are not allowed. Therefore, we call a potential function π *feasible*, if $\text{len}_\pi(u,v) \geq 0$ holds for all paths from u to v and arbitrary $u, v \in V$. It can be shown [GH05] that finding a shortest path in G is equivalent to finding a shortest path in G_π.

In [GH05] it is further proven that for a shortest s-t-path query using a *lower bound* from each node v to the target node t yields a feasible potential function. Note that the potential function is not needed to be fixed for all queries, but can be, as it is in our case, dependent on the s-t-query. In the original A* algorithm the lower bound used is simply the direct geographic distance (beeline) $\text{dist}_{\text{geo}}(v)$ from any node v to t. This works only as long as the metric in the graph is also geographical distance since then $\text{dist}_{\text{geo}}(v)$ is guaranteed to be a lower bound of any path from v to t. In our case this cannot be applied since we use travel time as metric. The A*-based ALT algorithm, however, introduces landmarks which are a (limited) set of dedicated nodes, from which lower bounds are computed with the aid of the triangle inequality.

Landmarks. For exact lower bounds we could precompute the distances between all pairs of nodes. However, this is far too expensive regarding both preprocessing time and space consumption. Thus, we only select a small subset $\mathcal{L} \subset V$ of landmarks and



(a) Visualizing Equation (5.2) **(b)** Visualizing Equation (5.3)

Figure 5.2.: The two pictures show the application of the triangle inequality according to the Equations (5.2) and (5.3). The 'detour' is drawn in light while the 'shortcut' is drawn in bold.

compute an exact distance table for all nodes v to/from every landmark $\ell \in \mathcal{L}$. The number of landmarks is usually set between 16 and 64. Because distances on G form a metric, the following instances of the triangle inequality hold.

$$\operatorname{dist}(v,t) + \operatorname{dist}(t,\ell) \geq \operatorname{dist}(v,\ell) \quad \text{and} \tag{5.2}$$

$$\operatorname{dist}(\ell,v) + \operatorname{dist}(v,t) \geq \operatorname{dist}(\ell,t). \tag{5.3}$$

Here $\operatorname{dist}(u,v)$ for two nodes $u, v \in V$ denotes the length of the shortest path from u to v in G. See also Figure 5.2 for a visualization of the matter. Resolving both equations to $\operatorname{dist}(v,t)$ (which is the value we are trying to bound), we obtain the feasible lower bound function

$$\pi_\ell(v) := \max\left\{ \operatorname{dist}(v,\ell) - \operatorname{dist}(t,\ell), \operatorname{dist}(\ell,t) - \operatorname{dist}(\ell,v) \right\} \leq \operatorname{dist}(v,t). \tag{5.4}$$

The best lower bound π can be obtained by using the landmark yielding the greatest lower bound according to

$$\pi(v) := \max_{\ell \in \mathcal{L}} \max\left\{ \operatorname{dist}(v,\ell) - \operatorname{dist}(t,\ell), \operatorname{dist}(\ell,t) - \operatorname{dist}(\ell,v) \right\}. \tag{5.5}$$

With $\operatorname{dist}(v,\ell)$ and $\operatorname{dist}(\ell,v)$ precomputed for each landmark $\ell \in \mathcal{L}$ and every node $v \in V$, the reduced cost graph G_π is then computed implicitly by altering the key of v in the priority queue to $\operatorname{dist}(s,v) + \pi(v)$.

Preprocessing. Preprocessing is done in two steps. First, a 'good' set of landmarks $\mathcal{L}$ is selected from V and, second, the distance table for $\mathcal{L}$ is computed. While the latter is straightforward and can be solved by running instances of Dijkstra's algorithm, the first is a non-trivial task. A 'good' landmark should yield a preferably high lower bound during the query for as many s-t-queries and for as many nodes v as

possible. In [GH05, GW05, DSSW06] evaluations of different techniques for selecting landmarks (on road networks) are presented, from which we use Avoid [GH05] and MaxCover [GW05]. While the quality of MaxCover landmarks is better (in the sense that they yield better lower bounds and, thus, faster query times) than that of Avoid landmarks, the first are harder to compute and, thus, not feasible when selecting more than 16 landmarks on large graphs as preprocessing times are getting too high. For that reason, we use MaxCover when selecting up to 16 landmarks and Avoid otherwise.

Query. The adaption of the query algorithm is easy. The only difference to Algorithm 3 on page 46 is that instead of using $\text{dist}_s(v)$ as keys in the priority queue, we use the cost reduced distance function $\text{dist}_s(v) + \pi_(v)$. However, previous experiments revealed that computing the lower bound with respect to *all* landmarks (cf. Equation (5.5)) produces too much overhead during the query. For that reason, we apply Equation (5.5) only on a subset $\mathcal{L}_{\text{active}} \subseteq \mathcal{L}$ of *active* landmarks. We usually restrict the cardinality of $\mathcal{L}_{\text{active}}$ to 2. The choice which landmarks are active depend on the query and are determined in the beginning using $\pi_\ell(s)$. Furthermore, every k iterations of the algorithm we update the set of active landmarks by rechecking which landmarks yield the best lower bound for the currently settled nodes.

Time-Dependency. The adaption of ALT to time-dependent networks only requires an adjustment on the preprocessing side of the algorithm. The query is left unchanged (except of the aspect that we evaluate the edge weights according to the departure time τ), as long as the potential function is guaranteed to be feasible for all possible times of day τ. For that reason, the distance table for the set $\mathcal{L}$ of selected landmarks is computed on the *lower bound graph* $\underline{G}$. So $\text{dist}(v, \ell)$ and $\text{dist}(\ell, v)$ only yield the distance from v to ℓ (and vice versa) for the best possible time τ over the day. However, since for all other times of day τ', it holds that $\text{dist}(v, \ell, \tau') \geq \text{dist}(v, \ell)$ and $\text{dist}(\ell, v, \tau') \geq \text{dist}(\ell, v)$, the computed distances can be regarded as a 'lower bound of a lower bound'. Depending on the difference between the lower and upper bounds of the edge weight functions in G, the quality of the precomputed distances can become very poor, thus, the ALT algorithm does not perform as good in time-dependent networks as in time-independent networks [NDLS08].

Adapting to Multi-Modal Routing. The adaption to multi-modal routing turns out easy. In a multi-modal graph $\boldsymbol{G}$ we observe that for every language L and every shortest s-t-path P conforming to L it has to hold that

$$\text{len}(P) \geq \text{len}(P_{\text{uni}}), \tag{5.6}$$

where $\text{len}(P_{\text{uni}})$ is the length of an uni-modal shortest s-t-path in $\boldsymbol{G}$ (i.e., we have no restriction due to a language L, or in other words, $L = \Sigma^*$, which allows arbitrary paths). In other words, applying constraints to shortest paths never yields shorter paths. Hence, the ALT algorithm can be adopted without any modifications. The preprocessing is done on $\underline{\boldsymbol{G}}$ yielding valid lower bounds for the distance tables. These lower bounds are generally worse, as for $\text{dist}(v, \ell)$ (and vice versa) we do not only use a lower bound regarding the departure time but also regarding the transportation modes. Thus, we expect another loss in quality regarding the potential functions which therefore decreases achievable speed-ups. On the bright side however, the query algorithm needs almost no modifications. We continue using the reduced cost function as priority queue keys as we did with the uni-modal versions of the algorithm.

Discussion. ALT is probably the most well-tempered speed-up technique, as it can be adapted very easily to time-dependent and multi-modal routing. Moreover, past experiments have shown that it is very robust to the input as it performs constantly well under various inputs [BDW07]. However, applying ALT as an uni-directional speed-up technique does not yield high speed-ups. Running a bi-directional setup with two independent ALT algorithms as described in Section 5.1.1, may lead to false results. Roughly speaking the main reason is that the forward and backward search do not use the same reduced costs on the edges. As a consequence, the node where the two search spaces meet is not necessarily on the shortest path. Although, in [GH05, GW05] a method is presented that adopts bi-directional search to the ALT algorithm, we omit further details at this point since we only use ALT as an uni-directional speed-up technique in this work.

5.1.3. Arc-Flags

Besides ALT, the arg-flag approach is another goal-directed speed-up technique for Dijkstra's algorithm. However, the underlying idea is different. While ALT tries to 'push' the search toward the target t by altering the priority of the nodes in the queue, Arc-Flags tries to find paths that can be pruned during the search, thus, yielding a smaller search space. This information is stored as flags on the edges (arcs) of the graph. The arc-flag approach is first introduced by Lauther [Lau97, Lau04]. Its key idea is the following insight.

Let $t \in V$ be a (arbitrary) node, then E_t denotes the set of all edges $e = (u, v) \in E$ where a shortest path to v starting at u uses the edge e. When computing a shortest s-t-path, Dijkstra's algorithm can be restricted to the edge-induced subgraph $G(E_t)$ of G. A proof of correctness can be found in [MSS+06].

Precomputing E_t for all $t \in V$ would require too much time. Hence, the node set V is *partitioned* into a family of r *cells* or *regions* $\mathcal{P} := \{R_1, \dots R_r\}$. Each node $v \in R_i$ is assigned its region-id by region : $V \to \{1, \dots, r\}$. Edge sets are now computed with respect to regions instead of individual nodes. If $R_1 \subset V$ is a region, the set E_1 consists of all edges $e = (u, v) \in E$ where there *exists* at least one node $t \in R_1$ where a shortest path to t starting at u uses the edge e. On the other hand, for an edge e we say that the region R_i can be *reached* through e, if there is at least one node $v \in R_1$ where e is on a shortest path to v.

During preprocessing the sets E_i for every region have to be computed. The information whether an edge e is on a shortest path to region i is attached to the edges. The computation of the subgraph $G(E_i)$ for an s-t-query (where $t \in R_i$) is done implicitly during the query by ignoring edges that do not reach the target region E_i.

Preprocessing. Arc-Flags requires a two-step preprocessing. First, a partition $\mathcal{P}$ has to be computed. Then, for each cell $R_i \in \mathcal{P}$ the set E_i according to the definitions above has to be computed. This information is then stored as flag vectors to the edges of the graph where, in short, the i'th bit of the vector at e indicates whether the edge is important for at least one node in the i'th region.

Partitioning the Graph. Experiments show (see [HKMS06, MSS$^+$06]) that the type of the partition has a major impact on query performance of the Arc-Flags query algorithm. Many of the partitioning methods tested in [MSS$^+$06] are geometric, i.e., they require an embedding of the graph into the plane. However, the arc-flag approach does not necessarily require geometric information by itself. While in fact any partition works, a 'good' partition should have the following properties. First, the cells of the partition should be connected. This helps the goal-direction of Arc-Flags. Moreover, the number of boundary nodes (i.e., nodes that are incident to edges connecting different cells) should be low. The main reason is, as we see soon that a high number of boundary nodes implies a high preprocessing effort. It should be noted that the geometric partitioning methods in [HKMS06, MSS$^+$06] fulfill the first claim, while the second is not considered. Hence, in [HKMS06] multi-way arc-separator approaches for partitioning the graph are tested (see [KK98] for an example). These do not require geographical information attached to the nodes and compute the partition solely based on the structure of the graph. We omit further technical details at this point and only like to point out that from the tested partitioning methods, METIS [Kar07], PARTY [MS04] and SCOTCH [Pel07], the latter yields the most promising results for Arc-Flags.

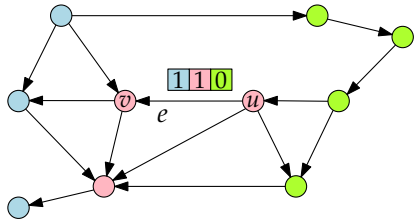


Figure 5.3.: A small graph partitioned into three cells (blue, red and green). The arg-flags vector is illustrated at the edge e. A shortest path from u targeting the green region never uses the edge e, thus the green flag of e can be set to `false`. However, for reaching the red and blue regions the edge e is important, thus, their flags are set to `true`.

Computing Arc-Flags. Given a partition $\mathcal{P}$, the second preprocessing phase involves the computation of the subgraphs E_i for each region $R_i \in \mathcal{P}$. For that reason, each edge gets a flag vector of length r assigned. For every edge e and every $i \leq r$, the i'th flag is set to `true` if and only if $e \in E_i$. These vectors on the edges are called *arc-flags* and indicate which regions can be reached by using the respective edge. Please also refer to Figure 5.3 for a small example illustrating the arc-flags vector.

The simplest and most naive way to compute arc-flags wrt. to region R_i is the following. First, we initialize the i'th flag on all edges except those edges with their tail inside R_i with `false`. Now we consecutively grow a full backward shortest path tree from every node $v \in R_i$, setting the i'th flag to `true` for every edge that is contained in the tree. Arc-flags once set to `true` are never changed back to `false`. To complete this approach for every region, we end up computing $|V|$ full backward shortest path trees in $\overleftarrow{G}$ which is too slow for large graphs (see also [Lau04]).

A faster method only uses boundary nodes. The key observation is that a shortest path with target region R_i has to enter the region at some point. Hence, it is sufficient to compute backward shortest path trees from the boundary nodes (instead of the nodes in the region). This decreases the preprocessing time significantly [Lau04]. However, preprocessing time is still rather high (several hours; even up to days on large graphs), which is the major drawback of the Arc-Flags approach. Thus in [HKMS06] another approach is presented using centralized shortest path trees which decreases preprocessing time significantly.

Query. The adaption of Dijkstra's algorithm to Arc-Flags is easy. Line 10 of Algorithm 3 on page 46 is altered such that only edges e with the i'th arc-flag enabled are considered where i is the region-id of the target node. Thus, the search is implicitly restricted to the edge induced subgraph E_i. Since Arc-Flags (in contrast to ALT) is goal-directed in the sense that it prunes paths instead of only 'guiding' the search, we

have to be careful about not losing the shortest paths by cutting it off. This turns out as the main difficulty when adapting Arc-Flags to multi-modal routing.

Time-Dependency. Augmenting Arc-Flags to cope with time-dependency is more difficult than one might anticipate. If we set ourselves the claim that the arc-flag vectors should not encode time-dependency, we have to enable flags if the respective edge is important for the target region at least once over the day. There are two approaches we describe in brief.

Exact Arc-Flags with Profile Queries. To account for time-dependency, we use the same preprocessing method as described in the previous paragraph. However, instead of growing simple backward shortest path trees, we do a backward profile search in $\overleftarrow{G}$ from all boundary nodes to every node in the graph. The labels attached to the nodes then indicate which incident edges are important for which time of day. Enabling every arc-flag on edges that are at least once over the day important for the respective target region, thus, yields correct time-dependent arc-flags.

This approach has a major drawback. Since computing profile queries involves a label correcting algorithm (cf. Algorithm 4 on page 50) which is significantly slower than a time-independent Dijkstra, makes this approach unfeasible for large graphs.

Approximating Arc-Flags. This drawback can be remedied by approximating the arc-flags. Note that enabling too many flags does not violate the correctness of the query algorithm. At most, it degrades the query performance by opening unnecessary arcs. In [Delo8] two approaches for approximating the arc-flags are introduced. The first one manages without profile queries. Instead, two time-independent backward shortest path trees are grown from each boundary node b of region R_i. One tree on the upper bound graph $\overleftarrow{\overline{G}}$ and another on the lower bound graph $\overleftarrow{\underline{G}}$. This yields two distance labels $\overleftarrow{\overline{\text{dist}}}(v)$ and $\overleftarrow{\underline{\text{dist}}}(v)$ for every node v in the graph. An arc-flag for an edge $e = (u, v)$ toward region i is then set to `true` if the inequality

$$\overline{f}_e + \overleftarrow{\overline{\text{dist}}}(v) > \overleftarrow{\underline{\text{dist}}}(v) \tag{5.7}$$

does not hold. This approach is fast, however, there might be too many enabled edges that are not essential for shortest paths toward region R_i.

Adaption to Multi-Modal Routing. While the adaption of Arc-Flags to time-dependency can be done (though with the penalty of higher preprocessing times) incorporating multi-modality into this approach turns out hard. While computing the partition

can be augmented easily, computing arc-flags is not straightforward. This is due to the fact that the automaton which is used during the query is not known beforehand.

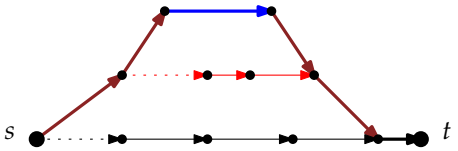
The first approach to tackle this problem might be to use a specific automaton during preprocessing. While this yields correct shortest-path queries, the major disadvantage of this approach is that the computed arc-flags are fixed to the respective automaton used during preprocessing. As a consequence, the query algorithm cannot be used on other automata. By these means we basically obtain a uni-modal shortest path problem, which makes this approach uninteresting. An intuitive augmentation is to compute arc-flags that work on whole classes of automata. In Section 4.3.2 on page 57 we introduced a reasonable class of hierarchical automata. However, we show with the aid of some examples that it is not clear how to compute arc-flags that allow correct queries on all instances of hierarchical automata.

Fixing the Automaton. Let $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ be an arbitrary multi-modal graph and $\mathcal{A} = (\Sigma, Q, \delta, S, F)$ a finite automaton. The Arc-Flags preprocessing step is modified as follows. The partition $\mathcal{P} = (R_1, \ldots, R_r)$ is computed on $\boldsymbol{G}$ (without knowledge of the automaton). This is important, as the exact final state is not known in advance when stating a multi-modal query. Hence, equivalent product-nodes consisting of different final states need to be in the same cell. By computing the partition on $\boldsymbol{G}$, this is achieved implicitly.

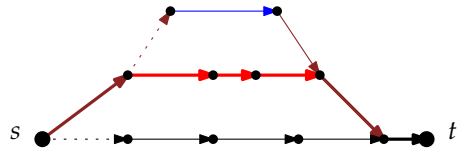
Now let R_i be an arbitrary region. Regarding the approach where we grow backward shortest path trees from all nodes of the region, we alter the method as we do not use simple nodes $v \in R_i$, but grow backward shortest path trees from all product-nodes (v, q_f) where $v \in \boldsymbol{V}$ and $q_f \in F$. Note that we can restrict ourselves to final states, as a multi-modal query targeting some node $t \in R_i$ has to end in a final state of the automaton. Then we set the i'th flag of an edge $e = (u, v) \in \boldsymbol{E}$ to `true` if for arbitrary $q_1, q_2 \in Q$ the product-node (v, q_1) is a predecessor of (u, q_2) on the backward shortest path tree in $\overleftarrow{\boldsymbol{G}}$. It shall be noted that this has the effect of merging arc-flags in the (implicit) product network $G^{\times}$ to $\boldsymbol{G}$. Thus, arc-flags on edges in the product network that are not required to be open could still be enabled.

We deemed the approach for computing arc-flags by growing shortest path trees from all nodes too expensive. Thus, we improved on the approach by only using the boundary nodes of a region as source nodes for computing backward shortest path trees. We can also adapt this improvement to multi-modal arc-flags. However, we now have to consider all states (not only final ones) of the automaton. Thus, for computing arc-flags for region R_i, we run backward shortest path trees for all boundary product-nodes (v, q) of the specific region, where $q \in Q$ is an arbitrary state of the automaton.

The multi-modal query algorithm is then used as shown in Algorithm 5 on page 55,



(a) General hierarchical automaton: Car, railway and flight network.



(b) Restricted hierarchical automaton: Car and railway network.

Figure 5.4.: This figure illustrates the challenge of computing arc-flags that work with a whole class of automata during query.
Both figures show the same multi-modal network with car edges (black), railway edges (red) and flight edges (blue) with arc-flags set with regard to the target node t. The shortest s-t-path obtained during preprocessing is drawn in bold. The arc-flags to the left are computed with the 'general' hierarchical automaton (all networks allowed), while the flags to the right are computed on a restricted automaton (only car and railways). Moreover, edge weights are defined in the way that it is always beneficial to use the highest possible network in the hierarchy for an s-t-query.
However, the left figure shows that using the restricted (car and railways) automaton during Arc-Flags query, we do not find a path from s to t since both the railway and car networks are pruned (dotted edges). Using the general automaton on the right, we will find a path, however, it is not the shortest as the flight network is pruned (which should be used due to the automaton).

with the exception that in line 11 only those edges $e = (v, w)$ are considered, for which the i'th arc-flag is set to `true` when performing an s-t-query with $t \in R_i$. Moreover, the automaton $\mathcal{A}$ is no longer part of the input. Instead, we use the same automaton as during preprocessing for every query. For that reason, Algorithm 5 always operates on the same (implicit) product network $G^{\times}$. Therefore, we believe this approach is too restrictive for accelerating general multi-modal queries.

Classes of Automata. To remedy this disadvantage, an intuitive augmentation would be to compute arc-flags that allow the usage of whole classes of automata (instead of only one automaton) during the query. In Section 4.3.2 the class of hierarchical automata was introduced which resemble a reasonable restriction on languages for multi-modal routing (see also Figure 4.4 for an example). However, it turns out that computing arc-flags that yield correct shortest paths for arbitrary automata during query is non-straightforward.

If we use the same approach for computing arc-flags as in the previous paragraph (i.e., using a multi-modal algorithm to grow the shortest path trees), it is not clear what automaton $\mathcal{A}$ from the respective class should be used. Figure 5.4a shows an example where arc-flags on a small graph are computed with the automaton of Figure 4.4.

Using these flags for a s-t-query together with an automaton that only allows the car and railway parts of the hierarchy, does not even find a path from s to t, because both the railway and the car subnetworks are cut off. They are never beneficial if the flight network can be used. On the other hand, as shown in Figure 5.4b, the reverse is not correct either. Using the restricted hierarchical automaton (car and railways) for computing the arc-flags prunes the flight-network, leads to incorrect shortest paths when a broader automaton also allowing flights is used.

Basically, the question we ask for is if there is a hierarchical automaton that can be used for arc-flags preprocessing such that every other automaton from the class of hierarchical automata can be used to compute correct shortest path using the multi-modal Arc-Flags-Dijkstra algorithm. Formally:

Problem 1. *Let $\mathfrak{A}$ be a class of automata (e.g., the class of hierarchical automata). Find a well-founded relation $\succ$ on $\mathfrak{A}$ such that the 'minimal' automaton $\mathcal{A}_0 \in \mathfrak{A}$ defined by*

$$\forall \mathcal{A} \in \mathfrak{A} : \mathcal{A} \succ \mathcal{A}_0 \tag{5.8}$$

can be used for Arc-Flags preprocessing such that every automaton $\mathcal{A} \in \mathfrak{A}$ yields correct queries with respect to the arc-flags computed with $\mathcal{A}_0$.

If we instantiate this problem with the class of hierarchical automata, we have seen in Figure 5.4a that using the (intuitively) most 'general' automaton of that class does not work. It is obvious that using restrictive automata like in Figure 5.4b also does not work as we always prune paths for modes of transportation that are not allowed by those means. Hence, we presume that for the class of hierarchical automata Problem 1 has no solution. Thus, the problem can be augmented in order to ask whether there exists a (non-trivial) class of automata that has a solution to Problem 1:

Problem 2. *Given the multi-modal arc-flags approach, is there a non-trivial (i.e., containing more than one element) class of automata $\mathfrak{A}$, for which Problem 1 can be solved?*

We leave this question open for further research on the topic.

As a final note regarding multi-modal Arc-Flags, we like to recall that our multi-modal graphs are mixed of both time-independent and time-dependent edges. For that reason, we also need to apply the augmentations discussed in the paragraph about time-dependency when computing multi-modal arc-flags.

Discussion. Arc-Flags is a goal-directed speed-up technique in the sense that it *prunes* paths that do not lead toward the target. The graph is partitioned into r cells and for each cell R_i we compute the set of edges E_i that are on a shortest path toward a node in cell R_i. This information is stored as r-bit flag vectors along the edges indicating

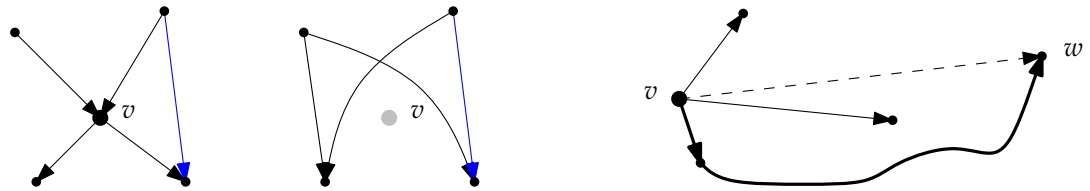
whether the resp. edge is important for the specific target cell. For uni-modal time-independent routing, this technique is a basic ingredient for one of the fastest speed-up techniques on road networks, SHARC-routing [BD08], CHASE and Transit-Node Routing with Arc-Flags [BDS+08, BDS+09]. SHARC has been augmented to routing in time-dependent road and railway networks in [Del08], still yielding very good results. However, it turns out that generalizing Arc-Flags further on multi-modal networks is very hard (if possible at all). While this question is an interesting topic for further (theoretical) research, we do not use Arc-Flags for our speed-up techniques in this thesis.

5.1.4. Contraction

The last basic ingredient we present is *contraction*. By contraction we mean some sort of reduction concerning the size of the graph, either, regarding nodes or edges. The former is called *node-reduction*, while the latter is referred to as *edge-reduction*. In road networks it turns out that contraction is a very powerful technique. Highway Hierarchies introduced in [SS05] and refined in [SS06a] is one of the first speed-up technique using contraction based methods. Contraction Hierarchies [Gei08, GSSD08] is a further development based on Highway Hierarchies, yielding a high-performance speed-up technique solely based on the concept of contraction. Moreover, contraction can be combined well with goal directed and bi-directional search. SHARC-routing (see [BD08]) combines contraction with Arc-Flags, while the Core-ALT algorithm (see [BDS+08]) combines contraction with the goal-directed search (ALT). In fact one of the fastest speed-up technique of today combines Contraction Hierarchies with Arc-Flags [Sch08a, BDS+08].

It turns out that contraction is also helpful to multi-modal routing (implying only marginal restrictions on the finite automata during queries). We describe both *node-reduction* and *edge-reduction* in this section. When both routines are applied only once, they yield a small graph called the *core*. The core-graph is significantly smaller than the original graph. It turns out that the query can be reduced to a many-to-many shortest path search on the core graph, where another speed-up technique can be applied orthogonally. Both our methods, Core-ALT and Core-Based Access-Node Routing make use of this approach. Another advantage by applying a speed-up technique only to the much smaller core graph is that preprocessing time and additionally required space can be reduced significantly.

Node-Reduction. The node-reduction when applied once, divides the graph into two parts: the *core* and the *component*. In the beginning all nodes are considered to belong



(a) Before. **(b)** After. **(c)** Edge-reduction on (v, w).

Figure 5.5.: Figures (a) and (b) illustrate the bypass operation during node-reduction. For each pair of incoming and outgoing edges at v, a shortcut is inserted. If the shortcut e is already contained in the graph (blue edge), the weight on the shortcut is set to the minimum weight of e and the weight of the edge that would be inserted at that place.
Figure (c): Illustration of the edge-reduction routine. The bold path from v to w is shorter than the edge $e = (v, w)$, thus, e can be deleted (dashed line).

to the core. We then consecutively *bypass* nodes, i.e., they get extracted from the core and eventually belong to the component of the graph, until no more nodes are bypassable. The criterion to which nodes are selected for bypassing as well as the criterion that indicates that no more nodes are bypassable are not discussed at this point. We go into detail regarding this issue when we introduce Core-Based Routing in Section 5.2. We only present the general concept of node-reduction here.

Let $G = (V, E)$ be a graph and $v \in V$ a node that belongs to the core of the graph. Then, for every incoming edge $e_{\text{in}} = (u, v)$ and every outgoing edge $e_{\text{out}} = (v, w)$ we insert a shortcut edge $e := (u, w)$ from u to w. The weight of e is set to $w(e) := w(e_{\text{in}}) + w(e_{\text{out}})$. If the insertion of e would lead to a multi-edge from u to w, we set the weight of the already existing edge $e' = (u, w)$ to the minimum $w(e') := \min\{w(e), w(e')\}$. Finally, the node v and all its incident edges are deleted from the graph. Note that the node-reduction routine preserves correct distances between two arbitrary core nodes. The sub-figures (a) and (b) in Figure 5.5 illustrate the bypass operation.

The obtained core graph is denoted by $G_{\text{core}} = (V_{\text{core}}, E_{\text{core}})$, while the component is defined as $G_{\text{comp}} = (V_{\text{comp}}, E_{\text{comp}})$ where $V_{\text{comp}} := V \setminus V_{\text{core}}$ and $E_{\text{comp}} = E \setminus E_{\text{core}}$.

Edge-Reduction. Performing the node-reduction potentially inserts many shortcuts into the core that may turn out unnecessary for preserving correct distances. Therefore, the edge-reduction works as follows. For each core node $v \in V_{\text{core}}$ and every edge $e = (v, w)$ we perform a shortest v-w-path query. If $\text{dist}(v, w) < w(e)$ (which is common for far shortcuts), the edge e is removed from the core. This can be accelerated by growing a shortest path tree from the node v using Dijkstra's algorithm and stopping as soon as all neighbors of v have been settled. We now have the distances $\text{dist}(v, w)$ for all neighbors w of v computed at once and can proceed with removing

all unnecessary outgoing edges of v at once. Note that applying the edge-reduction also preserves correct distances between core nodes in G_{core}. Figure 5.5c illustrates the edge-reduction regarding a single edge (v, w).

Preprocessing. The preprocessing of our contraction routine consists of applying node-reduction once until no more nodes are bypassable. Afterward, performing the edge-reduction on the core graph G_{core} is performed. For the sake of simplicity, the query algorithm uses only one graph. The graph $G := G_{\text{core}} \cup G_{\text{comp}}$ is obtained by unifying the core graph with the component. Core nodes are thereby marked with a flag.

Query. While the query algorithms of the previous techniques turned out easy, our contraction routine requires a more complicated query algorithm. As input we are given a graph $G = (V, E)$ with designated core-nodes. For an s-t-query the algorithm works in two phases. The first phase operates on the component part of the graph, while the second operates only on the core.

Phase one instantiates a *bi-directional* search on the component of G. This is achieved by not relaxing edges that are contained in the core (i.e., edges $e = (u, v)$ for which both u and v have the core flags set). Note that by these means we in fact settle core nodes, we just abort the search as soon as they are first hit (if either s or t are core nodes, the forward resp. backward search terminates immediately). The set S of core nodes that are hit by the forward search is called set of *core-entry-nodes*, while the set T of core nodes hit by the backward search is called the set of *core-exit-nodes*. Phase one terminates if one of the following conditions holds.

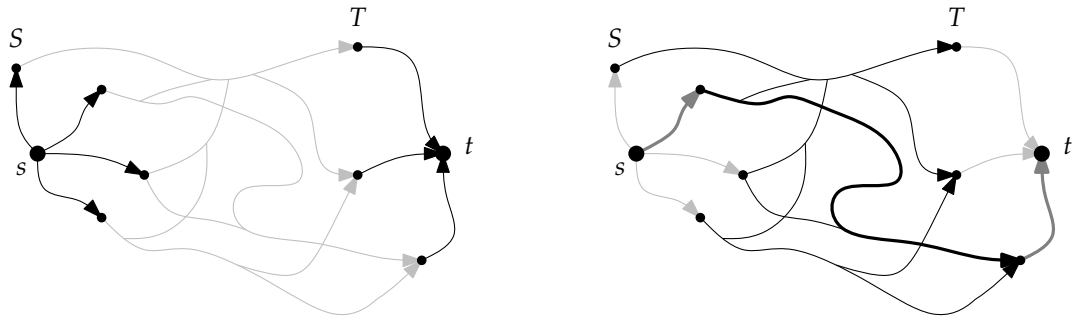
1. Both, the forward and backward priority queues are empty.
2. There has been a s-t-path P found for which it holds that

$$\text{len}(P) \leq \text{dist}(s, v_{\text{entry}}^{\text{min}}) + \text{dist}(v_{\text{exit}}^{\text{min}}, t), \tag{5.9}$$

   where $v_{\text{entry}}^{\text{min}} \in S$ denotes the core-entry-node with minimal distance from s in G, while $v_{\text{exit}}^{\text{min}} \in T$ denotes the core-exit-node with minimal distance from t in $\overleftarrow{G}$.

If phase one is aborted due to the second condition, we can stop the query and output the computed path P as shortest s-t-path. In this case, the shortest path solely uses nodes of the component.

So for the rest of this paragraph we assume that no path P in the component has been found by phase one. In this case, phase two of the algorithm is instantiated with a many-to-many S-T query only relaxing edges contained in the core. However, the



(a) Phase I. **(b)** Phase II.

Figure 5.6.: Illustration of the Core-Based Routing algorithm. Phase one (left) conducts a bidirectional search until all core-entry resp. core-exit-nodes have been reached. In phase two an S-T-query is performed on the core graph (right). The shortest path is then combined by taking from all s-T-t-paths the one having minimum length.

forward (and backward) queues are re-filled such that the initial keys are set to the distances (from s resp. t) computed in phase one. The algorithm used in phase two is not specified. Thus, contraction yields a modular design that allows combination with an arbitrary speed-up technique applied on the core.

The final s-t-path is then combined by determining the minimal s-T-t-path $P_{s,v,t}$ for every node $v \in T$ where the length is computed by

$$\text{len}(P_{s,v,t}) := \text{dist}(s,v) + \text{dist}(v,t). \tag{5.10}$$

Figure 5.6 illustrates the query algorithm.

Although the correctness of the query algorithm is not as obvious as with the other basic ingredients, we omit an extensive proof of correctness and like to refer the reader to [BDS+09] for further details.

Time-Dependency. The adaption of our contraction method to time-dependent routing is straightforward, however, comes with the penalty of high preprocessing times and space consumption.

Preprocessing. Regarding node-reduction the only difference is that we have to process travel time functions instead of constant weights. Thus, when inserting a shortcut $e = (u,w)$ for two edges $e_{\text{in}} = (u,v)$ and $e_{\text{out}} = (v,w)$ the function f_e is computed by $f_e := f_{e_{\text{in}}} \oplus f_{e_{\text{out}}}$. If an edge $e' = (u,w)$ already exists in the graph the new function on e' is computed by $f_{e'} := \min\{f_e, f_{e'}\}$. Besides being slower to process functions instead of constant weights, both the link- and merge-operation may increase the number of

interpolation points and, thus, lead to high space consumption of the precomputed data. This is especially true for road networks (see [Delo8]).

The time-consuming part is the edge-reduction routine. It is no longer sufficient to compute simple shortest path trees to determine for a node v which edges to its neighbors w may be deleted. Instead, we have to compute profile queries. An edge $e = (v, w)$ may then only be deleted if $f_w < f_e$ holds, meaning the computed path to w is faster than using the edge directly for all times of day. As we worked out in Section 4.2.2, computing profile queries is much slower than computing time queries.

Query. Adapting the query algorithm to time-dependency requires some careful modifications. Since phase one of the query uses bi-directional search, we encounter the same problem as we did with plain bi-directional search: The exact arrival time at the target node t is not known in advance. Hence, the algorithm is modified as follows.

Phase one, again, consists of a bi-directional search on the component graph (i.e., edges that belong to the core are not relaxed). Like drawn out in Section 5.1.1 the forward search is conducting a time query with departure time τ at s, while the backward search is time-independent on the backward lower bound graph $\overleftarrow{\underline{G}}$. However, the stop criteria differ from the time-independent algorithm. Phase one is now halted if one of the following criteria holds.

1. Both, the forward and backward priority queues are empty.

2. If S is the set of core-entry- and T the set of core-exit-nodes, the search is halted if $S \cap T \neq \emptyset$, i.e., there is at least one core node that is discovered by both searches. In this case, a standard uni-directional s-t-time query with departure time τ (on whole G) is started and the shortest path outputted.

In the case that phase one is stopped by the first criterion, the core algorithm is initialized—in the same manner as the time-independent version—with the sets S and T. However, the forward search is allowed to 'break out' of the core at the core-exit nodes T. Formally, as soon as the forward search settles a node $v \in T$, the algorithm is also allowed to relax edges outside the core. Thus, the shortest s-t-path is found directly by the forward search, rendering the step of finding the minimal path over all exit-nodes T due to Equation (5.10) unnecessary for this case. Again, we omit a detailed proof of correctness and refer the reader to [DNo8].

Adaption to Multi-Modal Routing. Adapting contraction to multi-modal routing entails some hurdles one should be aware of. For the sake of clarity, we discuss the adaption to multi-modal routing for every step during contraction separately.

Node-Reduction. When we insert a shortcut $e = (u, w)$ due to bypassing a node v, the edges (u, v) and (v, w) are linked into one edge. When we look at this from the perspective of the automaton, walking along one edge of the network implies exactly one transition in the automaton. So, when two edges are linked into one edge, an automaton during query is only able to perform one transition where it would have performed two in the original graph. Thus, in general, bypassing a node destroys the paradigm of being able to use any automaton for the query.

However, we can counteract that problem by restricting the contraction routine to nodes v satisfying the following property. For all incoming edges e_{in} to v and all outgoing edges e_{out} from v it must hold that they have the same edge labels. In that case, we only 'unify' transitions having the same label (symbol in the automaton). While we theoretically still lose the ability to use any automaton for the query algorithm, we retain the possibility to use every automaton where a transition with respect to some label σ can be repeated arbitrarily often. In Section 4.3.2 on page 57 we worked out that every reasonable automaton fulfills this property. Thus, we conclude node-reduction can be reasonably augmented to multi-modal routing.

Edge-Reduction. The edge-reduction routine has to be altered for the same reasons node-reduction had to be restricted. If v is a node belonging to a certain network type T, then the shortest path tree grown from v has to be restricted to that particular network T, i.e., the shortest path algorithm is only allowed to settle nodes labeled by T. In this case, it is guaranteed that a shortest path from v to one of its neighbors w (not using the edge $(v, w) =: e$ directly) is only composed of edges with the same label as e. Thus, it is safe to remove the edge e from the graph. Note that if $\text{label}(v) \neq \text{label}(w)$, then w is never settled by the shortest path algorithm and, thus, e is never removed. If performed with this constraint, the edge-reduction routine has the exact opposite effect of node-contraction. Removing an edge e from v to w keeps the distance from v to w correct but expands the number of subsequent $\text{label}(e)$-labeled transitions in the automaton regarding the segment from v to w on the path.

Query. Concerning the query algorithm we are set with the standard augmentation of replacing nodes by product-nodes which are computed implicitly. Thus, phase one yields two sets S and T of product-nodes, which are then re-inserted into the forward and backward queues with their proper distances. The phase two algorithm on the core can be chosen to be any many-to-many multi-modal routing algorithm. The augmentations of bi-directional routing to multi-modal networks (cf. Section 5.1.1), of course, also apply here for the phase one part of the query.

Discussion. Contraction is a basic ingredient for the best known speed-up techniques on uni-modal graphs as of today. The result after preprocessing is a graph that contains a contracted subgraph called the core. The query algorithm is more complex than with the other basic ingredients. It consists of two phases: The first phase runs a bi-directional search on the component until all entry- and exit-points to the core are found. The second phase then uses an arbitrary algorithm operating solely on the core graph. The main strength of contraction is that in practice the first phase terminates very quickly and because of the very small size of the core, phase two is restricted to a small node subset of the original graph, thus, accelerating the query [Sch08a]. Furthermore, our core-based approach of contraction allows for an arbitrary speed-up technique on the core.

Contraction can be augmented to time-dependency, though, with the penalty of higher preprocessing space and time. Furthermore, on road networks, contraction increases the complexity of the travel time functions on the edges [Del09a], thus, inflicting a penalty on the query side of the algorithm. The generalization to multi-modal networks is possible but we have to be careful as to which nodes and edges are allowed for bypassing and deletion. Roughly speaking, if we prohibit the application of the contraction routines over 'borders' of the different network types, we are still able to use all reasonable automata during the query

In the next section we refine our contraction routine on multi-modal networks yielding a robust technique we call Core-Based Routing. The main idea is to restrict contraction to the time-independent road network which makes up most part of our multi-modal graphs. Thus, the railway and the flight networks are completely contained in the core.

5.2. Core-Based Routing

In this section we present Core-Based Routing which is a pure multi-modal speed-up technique based on contraction. While in the previous section we already mentioned how contraction can be augmented to both time-dependency and multi-modality in general, Core-Based Routing as we present it here uses a slightly different approach.

We like to remind that our multi-modal networks are composed of a time-independent road network containing both foot and car edges as well as time-dependent railway and flight networks (cf. Section 3). When we introduced contraction in Section 5.1.4 we worked out that the adaption to time-dependent scenarios comes with the drawback of high preprocessing times (linking and merging of functions instead of processing constants and using profile queries instead of shortest path trees for computing the edge-reduction) and a more complicated query algorithm.

However, the largest portions of our multi-modal graphs are made up of the time-independent road network. For example a graph of Europe containing road, railway and flight networks contains 30 722 060 nodes from which 30 203 343 belong to the road network (cf. Section 6.1). Thus, we remedy the disadvantages of time-dependent contraction by restricting preprocessing to the road network. Therefore, the core contains the non-contracted public transportation networks as well as (contracted) parts of the road network, while the component consists only of nodes from the road network. Hence, we are able to use the time-independent version of the query algorithm (see previous section). Although the arrival time of the target is not known in advance, we are able to apply bi-directional routing on the component since the distances in the road network are independent from the arrival time of the target. The time-dependent part of the graph is fully contained in the core. Our basic core-based router uses a simple uni-directional time-dependent multi-modal DIJKSTRA algorithm during the core part of the query (cf. Algorithm 5).

In the subsequent sections we are always given a multi-model graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ with labeled edges and nodes. Furthermore, we assume that our multi-modal graphs contain a road network.

5.2.1. Preprocessing

Preprocessing of Core-Based Routing is based on the application of the node-reduction and edge-reduction routines as introduced in Section 5.1.4. At first, node-reduction is applied once and afterward the edge-reduction kicks in (also once) to remove unnecessary edges. Instead of extracting the core $\boldsymbol{G}_{\text{core}}$ and merging it with the original graph $\boldsymbol{G}$ in the end, we only operate on $\boldsymbol{G}$ the whole time. Thereby, nodes that belong to the core are marked by a flag $\text{core} : V \to \{\texttt{true}, \texttt{false}\}$. Although, we do not extract the core explicitly, we refer to the set of nodes $v \in \boldsymbol{V}$ having $\text{core}(v) = \texttt{true}$ as $\boldsymbol{V}_{\text{core}}$. Furthermore, edges that are inserted during the node-reduction are also marked with a flag so they can be identified as shortcuts.

Node-Reduction. We would like to remind that the node-reduction is made up of consecutive bypass operations. Let $v \in \boldsymbol{V}_{\text{core}}$ be a node that is not yet bypassed. Then applying the bypass operation on v yields shortcuts $e = (u, w)$ being inserted for every incoming edge $e_{\text{in}} = (u, v)$ and outgoing edge $e_{\text{out}} = (v, w)$. Since we want to restrict ourselves to the road network, the bypass operation is only applied if the following conditions hold.

1. The node v must be part of the road network, thus $\text{label}(v) = \texttt{ROAD\_NODE}$ must hold.

2. All neighbors of v must belong to the road network, thus label$(w) =$ `ROAD_NODE` must hold for all neighbors of v. Note that this implies that all edges incident to v are labeled as `ROAD_EDGE`.

3. All incident (road) edges to v must be open for the same modes of transportation (car/foot). These are either car and foot, only cars or only foot.

4. For all pairs of edges $(u, v), (v, w)$ and (u, w) in the graph, the edge (u, w) must also be open for the same modes of transportation (car/foot) as (u, v) and (v, w).

All newly inserted edges get their shortcut-flag enabled. We do not set this flag for edges $e \in \boldsymbol{E}$, even though we might change the weight of e. Since our preprocessing is done in-place on $\boldsymbol{G}$, we do not delete bypassed nodes. Instead, for a node $v \in \boldsymbol{V}_{\text{core}}$ we set core$(v) =$ `false`. Furthermore, all incident edges e to v that have their shortcut flag set are not contained in $\boldsymbol{E}$ originally. Thus, they can be removed when v is bypassed.

Order of Processing. In Section 5.1.4 we mentioned that we stop the node-reduction routine as soon as no more nodes are bypassable. Furthermore, experiments have shown (see [GSSD08]) that the order in which nodes are bypassed influences the quality of the contracted core graph regarding the number of edges. For that reason, we use a priority queue to maintain the order in which nodes are bypassed. Nodes with a low priority are bypassed first.

To determine if a node is *bypassable*, we examine the number of shortcuts (#shortcuts) that would be inserted if v would be bypassed. Note again that we only count the number of newly inserted edges. (Shortcut-) edges that would only be updated do not contribute to #shortcuts. The node v is bypassable if the following equation holds:

$$\#\text{shortcuts} \leq c \cdot \big(\deg_{\text{in}}(v) + \deg_{\text{out}}(v) \big). \tag{5.11}$$

Hereby, $\deg_{\text{in}}(v)$ and $\deg_{\text{out}}(v)$ denote the number of incoming resp. outgoing incident edges at v. Moreover, c can be specified as a tunable contraction parameter (higher values allow more aggressive contraction) and is usually chosen between 0.5 and 5.0.

Furthermore, it has to hold that

$$h(v) \leq H \tag{5.12}$$

where $h(v)$ denotes the number of hops on the hop-maximal shortcut that would be inserted. The number of hops on a shortcut is defined as the number of nodes that are bypassed by the shortcut. Note that when inserting shortcuts for two edges e_{in} and e_{out}, one of them may already be a shortcut inserted previously, thus the hop number

can grow arbitrarily big. The choice of H yields another parameter that controls contraction (higher valuies, again, allow for more aggresive contraction). The parameter H is usually chosen between 10 and 70.

If v is bypassable, i.e., both bypass criteria hold, the priority of a node $v \in \boldsymbol{V}_{\text{core}}$ is determined by the equation

$$\text{key}(v) := h(v) \cdot \frac{\#\text{shortcuts}}{\left(\deg_{\text{in}}(v) + \deg_{\text{out}}(v)\right)}. \tag{5.13}$$

So with regard to this equation we prefer the insertion of 'short' shortcuts. Moreover, nodes that would yield less shortcuts (relative to the number of neighbors) are preferred as well.

In the beginning the priority queue is filled with all nodes $v \in \boldsymbol{V}_{\text{core}}$ which are bypassable according to their priorities $\text{key}(v)$. Then, we extract the minimal element from the queue, bypassing the respective node. Since bypassing a node v can influence the key of its neighbors, we recompute their keys afterward and update the priority queue accordingly. Note that this may induce the removal of nodes from the queue prematurely. Our node-reduction routine finally terminates when the queue is empty, i.e., no more nodes are bypassable.

Edge-Reduction. The edge-reduction is applied as described in Section 5.1.4 with almost no modifications. Since we have to restrict ourselves to the subgraph only containing edges labeled by the same label as $\text{label}(e)$ when processing an edge $e = (v, w)$, we have to take some extra considerations. In Section 5.1.4 we optimized the edge-reduction by growing a shortest path tree from v to handle all outgoing edges $e = (v, w)$ at once. We like to retain this improvement, hence, we apply the edge-reduction to a node $v \in \boldsymbol{V}_{\text{core}}$ if the following conditions hold.

1. The node v is part of the road network, i.e., $\text{label}(v) = \texttt{ROAD\_NODE}$,
2. all outgoing edges $e = (v, w) \in \boldsymbol{E}_{\text{core}}$ are labeled by `ROAD_EDGE` and
3. they allow the exact same modes of transportation (foot or car). These are car and foot, car alone and foot alone.

Now let v be a node on which the edge-reduction can be applied. Then we grow a time-independent shortest path tree from v restricted on the time-independent road network, i.e., we do not relax edges that are not labeled as `ROAD_EDGE`. Furthermore, we restrict the computation to edges in the road network that are open for the exact same modes of transportation as the outgoing edges of v. If the edges e are open for either cars or pedestrians alone, we remove $e = (v, w)$ if $w(e) > \text{dist}(v, w)$. If, however,

the edges e are open for both modes of transportation, we have to grow two distinct shortest path trees: One using car weights, the other using foot weights.

We then only remove the edge $e = (v, w)$ from the graph if both inequalities

$$w_{\text{car}}(e) > \text{dist}_{\text{car}}(v, w) \quad \text{and} \quad w_{\text{foot}}(e) > \text{dist}_{\text{foot}}(v, w) \tag{5.14}$$

hold at the same time. By these means we ensure that the removal of the respective edges does not violate the distances between nodes in the graph. Furthermore, we ensure that the alternate path only contains edges labeled by the same modes of transportation as the deleted edge.

5.2.2. Query

Our query algorithm for Core-Based Routing is applied on a contracted multi-modal graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ with labeled nodes and edges. Furthermore, nodes belonging to the core are designated by the core flag. Our query algorithm performs multi-modal time queries (cf. Section 4.2.2), i.e., we are given source and target nodes $s, t \in \boldsymbol{V}$, a departure time τ at s and a regular language $L \subset \Sigma^*$. Furthermore let $\mathcal{A} = (\Sigma, Q, \delta, S, F)$ be the finite automaton accepting L.

As presented in the previous section, preprocessing on $\boldsymbol{G}$ is restricted to the road network, therefore the component consists of time-independent edges alone. For that reason, applying a bi-directional search algorithm restricted to the component of $\boldsymbol{G}$ yields correct shortest paths (see also Section 5.1.1 for a description of a bi-directional time-independent Dijkstra). Hence, the time-independent version of the query algorithm as described in Section 5.1.4 can be adopted with only a few modifications to account for multi-modality. The algorithm is made up of two phases. We recapitulate the query algorithm in the following and tailor it to our version of the contraction routine.

Phase I. Phase one consists of a bi-directional multi-modal time-independent search algorithm restricted to the component. The forward search starts at s in $\boldsymbol{G}$, while the backward search starts at t in $\overleftarrow{\boldsymbol{G}}$. Note that the priority queues are initialized with product-nodes (s, q_s) for all $q_s \in S$ and (t, q_f) forall $q_f \in F$ (see also the augmentation to multi-modal routing of bi-directional search in Section 5.1.1).

Both searches only relax edges which are not in the core, i.e., an edge $e = (u, v)$ where $\text{core}(u) = \texttt{true}$ and $\text{core}(v) = \texttt{true}$ is not relaxed. By these means we discover sets S and T of core-entry- and core-exit-product-nodes, respectively. Phase one terminates, as soon as either both priority queues are empty or we discover a shortest path which solely uses the component. Such a shortest path is found if the bi-directional

search on the component found and s-t-path P for which it holds that

$$\text{len}(P) \leq \text{dist}_s\left((v,q)^{\min}_{\text{entry}}\right) + \overleftarrow{\text{dist}}_t\left((v,q)^{\min}_{\text{exit}}\right), \tag{5.15}$$

where $(v,q)^{\min}_{\text{entry}}$ denotes the core-entry-product-node with minimal distance from s and $(v,q)^{\min}_{\text{exit}}$ the core-exit-product-node with minimal distance from t. $\overleftarrow{\text{dist}}$ denotes distances on $\overleftarrow{\boldsymbol{G}}$. In this case P is outputted as shortest path and the algorithm terminates. Otherwise, we continue with phase two.

Phase II. For phase two, we are given sets S and T of core-entry- resp. core-exit-product-nodes together with the distances of every node in S and T from the source s resp. target t. Phase two then consists of a many-to-many multi-modal time-dependent shortest path algorithm from S to T restricted to $\boldsymbol{G}_{\text{core}}$, i.e., edges $e = (u,v)$ with $\text{core}(v) = \texttt{false}$ are not relaxed by the algorithm. While in theory, any algorithm can be used, we use the standard multi-modal uni-directional DIJKSTRA algorithm for the basic version of Core-Based Routing (cf. Algorithm 5 on page 55). The forward queue from phase one can be re-used and the product-nodes $(v,q) \in S$ get re-inserted into the queue with key $\text{dist}_s\left((v,q)\right)$. Phase two terminates as soon as either the priority queue is empty, or all product-nodes from T have been settled.

Combining The Shortest Path. Phase two of the algorithm terminates with forward and backward distance labels for each of the core-exit-product-nodes $(v,q) \in T$. The final shortest path is then obtained through combining the paths from s to $(v,q) \in T$ with the path from (v,q) to t (which is resulting from the backward search of phase one). The node $(v,q) \in T$ is the node for which $\text{dist}_s\left((v,q)\right) + \overleftarrow{\text{dist}}_t\left((v,q)\right)$ is minimal.

5.2.3. Proof of Correctness

In this section we give a formal proof of correctness for our Multi-Modal Core-Based Routing algorithm.

Our proof is split into two parts. First, we prove that the contraction routine preserves distances on the core, i.e., for two nodes $u, v \in \boldsymbol{G}_{\text{core}}$, it holds that $\text{dist}(u,v)$ (wrt. to L) are not changed by contraction. In the second part we prove that the query algorithm then yields correct paths.

To prove that distances on the core are correct we first give a formal definition of a language L having the *variabl length property*.

Definition 8. *Let $L \subset \Sigma^*$ be a regular language where the alphabet Σ^* is the set of edge labels. Then L fulfills the* variable length property *if for every word $w \in L$ it holds that for every*

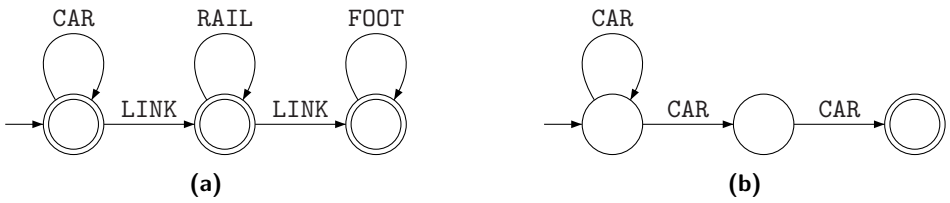


Figure 5.7.: Two automata. The one to the left fulfills the variable length property, the one to the right does not.

decomposition of w in $w = xyz$ where $x, z \in \Sigma^*$ and $y = \sigma^i$ for $\sigma \in \Sigma \setminus \{\texttt{LINK\_EDGE}\}$ and maximal $i \in \mathbb{N}$ the words

$$w' := x\sigma^k y \qquad \forall k \in \mathbb{N}_0 \tag{5.16}$$

are also contained in L.

If $\mathcal{A}$ is an automaton accepting L, we also say that $\mathcal{A}$ has the variable length property*.*

Figure 5.7 gives two examples of automata from which one has the variable length property and the other does not. Our contraction routine first applies node-reduction once and the edge-reduction routine afterward. During node-reduction we extract nodes from the core of $\boldsymbol{G}$. The remaining core subgraph is denoted by $\boldsymbol{G}_{\text{core}}$. Furthermore, let L be an arbitrary language fulfilling the variable length property.

Lemma 1 (Node-Reduction)**.** *Node-reduction preserves distances on $\boldsymbol{G}_{core}$ wrt. L.*

Proof. The lemma is proven by induction over the number of bypass operations. In the beginning, all nodes belong to the core and yet no changes have been made to the graph, thus, distances on the core are correct.

Now assume that distance are correct after a finite number of node bypass operations. We consider bypassing a node $v \in \boldsymbol{G}_{\text{core}}$, thus, we have to show that for all $x, y \in \boldsymbol{V}_{\text{core}} \setminus \{v\}$ distances wrt. to L are correct after v has been bypassed. For that matter, let P be the shortest path from any x to y. We distinguish the following cases.

Case I: The path P is not going through either v or any of the neighbors of v. Since the bypass operation only inserts resp. removes edges involving v and its neighbors, the path P is still contained in the graph after the bypass operation.

Case II: The path P is using v. Since v is extracted from the core, the path P no longer exists in the core. Thus, let $[u, v, w] \subset P$ denote the (not necessarily only) occurrence of v in P with its adjacent nodes along the path.

For that reason, time-independent road edges $e_1 := (u, v)$ and $e_2 := (v, w)$ have to exist in the graph before bypassing v. An edge $e_3 := (u, w)$ might exists, however, it has to hold that $w(e_3) \geq w(e_1) + w(e_2)$, otherwise the shortest path would go from u to w directly. Furthermore, these edges have to support the same modes of transportation (regarding car and foot), otherwise v would not be bypassable. Thus, the labels in the word in L corresponding to P can be described by σ^2 where $\sigma \in \{\texttt{CAR\_EDGE}, \texttt{FOOT\_EDGE}\}$.

Due to the construction rules of the bypass operation, either a new edge $e_{\text{new}} = (u, w)$ is inserted with weight $w(e_{\text{new}}) = w(e_1) + w(e_2)$. In this case a path P' in $\boldsymbol{G}_{\text{core}}$ having the same length as P can be constructed from P by substituting the subpath $[u, v, w]$ through $[u, w]$. The corresponding labels in P' are reduced from σ^2 to σ, thus, the resulting word corresponding to P' is still a member of L due to the variable length property.

If, on the other hand, no new edge e_{new} is inserted, there was already an edge e_3 in the graph. In this case, the weights of e_3 are set to the minimum of $w(e_3)$ and $w(e_1) + w(e_2)$ for each mode of transportation (car and foot). Because for the mode of transportation σ which is used along the respective segment of the path P, it holds that $w(e_1) + w(e_2) \leq w(e_3)$, the new weight of e_3 is set to $w(e_1) + w(e_2)$. Hence, we immediately obtain that the path P' constructed by the same rules as in the first case yields a path having correct distance and word wrt. to the language L.

Case III: The path P is not using v but an edge $e_3 := (u, w)$ where both u, w are neighbors of v in the sense that there are edges $e_1 := (u, v)$ and $e_2 := (v, w)$ contained in $\boldsymbol{E}_{\text{core}}$. Then it has to hold that $w(e_3) \leq w(e_1) + w(e_2)$. In this case the weight of e_3 is retained after the bypass operation and P is still a valid path in $\boldsymbol{G}_{\text{core}}$ after v has been bypassed.

Altogether, we obtain by induction that the distances with respect to L are preserved by our node-reduction routine. □

Lemma 2 (Edge-Reduction)**.** *Edge-Reduction preserves distances on $\boldsymbol{G}_{core}$ wrt. L.*

Proof. Again, we prove the lemma by induction. Assume that after n deleted edges the distances are still correct. Consider the edge-reduction applied to a node v where the edge $e := (v, w) \in \boldsymbol{E}_{\text{core}}$ is removed from the graph. Furthermore, assume that for two arbitrary nodes $x, y \in \boldsymbol{V}_{\text{core}}$ the path P uses the edge e by the mode of transportation $\sigma \in \{\texttt{CAR\_EDGE}, \texttt{FOOT\_EDGE}\}$ (otherwise nothing happens and we are set). By the definition of edge-reduction, there exists a path P' from v to w not using the edge e that has a shorter length than e. Furthermore, the word along P' is exactly σ^k whereas $k = |P'|$. Thus, we can substitute the edge e in P by P' yielding a new path P_{new}. This path still

forms a valid word wrt. to L since substituting σ through σ^k in the word conforms to the variable length property of L. However, it now holds that $\operatorname{len} P_{\text{new}} < \operatorname{len} P$ which is a contradiction to P being a shortest path in $\boldsymbol{G}_{\text{core}}$.

Thus, there is no shortest path P using the edge e and removing the edge e preserves distances wrt. to L in $\boldsymbol{G}_{\text{core}}$.

Because distances were correct before the edge-reduction routine is applied, we obtain that edge-reduction preserves all distances on $\boldsymbol{G}_{\text{core}}$ wrt. a variable length language L. □

Theorem 3 (Correctness of Core-Based Routing)**.** *Multi-modal Core-Based Routing is correct. Given a multi-modal graph $\boldsymbol{G}_C = (\boldsymbol{V}, \boldsymbol{E})$ after preprocessing $\boldsymbol{G}$, source and target nodes $s, t \in \boldsymbol{V}$, a departure time τ and a regular language L having the variable length property, applying the multi-modal Core-Based Routing algorithm on the contracted version $\boldsymbol{G}_C$ of $\boldsymbol{G}$ yields correct shortest paths from s to t at time τ wrt. the language L.*

Proof. Since during preprocessing of Core-Based Routing no nodes are deleted or inserted, we can identify the node sets $\boldsymbol{V}$ and $\boldsymbol{V}_C$. Now let

$$P = [(s, q_1), \ldots, (v_i, q_i), \ldots, (v_j, q_j), \ldots, (t, q_k)] \tag{5.17}$$

be a shortest path of length $k-1$ in $\boldsymbol{G}$. If all nodes in P belong to the component, then P is also found by our query algorithm in $\boldsymbol{G}_C$ by the correctness of time-independent bi-directional search. This also holds if only one product-node $(v, q) \in P$ belongs to the component since in that case no edge contained in the core is used along P.

Now let (v_i, q_i) and (v_j, q_j) for $i < j$ denote the first resp. last product-node in P that belong to the core. Let the sub-path of P between these product-nodes be denoted by P'. Then by Lemmas 1 and 2 we obtain that there is a path P_{core} of equivalent length solely using core-nodes. Moreover, the path P_{new} obtained by substituting P' through P_{core} conforms to the language L.

The paths $[(s, q_1), \ldots, (v_i, q_i)]$ and $[(v_j, q_j), \ldots, (t, q_k)]$ are found by phase one of the query algorithm. In this case it holds that (v_i, q_i) is a core-entry-product-node and (v_j, q_j) a core-exit-product-node. The path P_{core} is, thus, found by phase two of the query algorithm by the correctness of the multi-modal Dijkstra algorithm.

Altogether, we conclude that Multi-Modal Core-Based Routing yields correct shortest paths due to languages L fulfilling the variable length property. □

5.2.4. Discussion

In this section, we introduced our first tailored multi-modal speed-up technique. More precisely, we augmented the basic ingredient of contraction to multi-modal networks

in a way that preprocessing is fast and feasible. Moreover, the only loss of generality is that we are no longer able to use languages L that do not fulfill the variable length property during queries. However, this is only a small disadvantage, since we worked out that every 'reasonable' language L for multi-modal routing has this property anyway.

Regarding the query, we can use the time-independent uni-modal query algorithm of core-based routing as a template and only need to augment it with the necessary modifications required for multi-modality (product-nodes instead of simple nodes). Because the time-dependent parts of our graphs are fully contained in the core, the algorithm used for the 'inner' part of the query (phase two), can be chosen orthogonally. In this section we choose a simple multi-modal Dijkstra algorithm.

While on road-networks contraction can lead to extremely small core graphs of 1% size [Sch08a], these ratios are hard to achieve in our multi-modal scenario. This is due to two facts: First, the whole public transportation part of the core is not bypassed and therefore contained in the core. Furthermore, in our road networks, a lot of nodes are not bypassable as their incident edges are not homogenous regarding the open modes of transportation (foot and car). We observe that our graphs can be contracted down to approximately 15% of the original size. Also refer to Section 6.4 for experiments and figures on this matter.

Core-Based Routing by itself only yields mild speed-ups. However, in the next section we introduce uni-directional ALT on the core, resulting in a robust multi-modal speed-up technique with good performance.

5.3. Core-ALT

In this section we introduce Core-ALT (short: CALT) for time-dependent multi-modal networks. This speed-up technique is based on Core-Based Routing introduced in the previous section. We extend the Core-Based Routing approach by applying the uni-directional time-dependent multi-modal ALT algorithm as introduced in Section 5.1.2 on the core graph.

The idea to combine contraction with ALT is not new and has been done in [Sch08a]. This approach has then been adapted to time-dependency in [DN08] called TDCALT. In this section we augment Core-ALT further to multi-modal networks. In contrast to [DN08] our networks are not purely time-dependent. As worked out in the previous section, only our core has time-dependent parts, thus, we can continue to use the time-independent query algorithm on the 'outer' part of Core-Based Routing and only need to exchange the 'inner' algorithm by time-dependent ALT. Moreover, we use the uni-directional variant of ALT. This spares us the rather complicated query algorithm for

bi-directional TDCALT [DN08].

5.3.1. Preprocessing

Preprocessing for multi-modal Core-ALT is straightforward. Given a multi-modal graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$, we first compute the contracted graph $\boldsymbol{G}_C$ in the same way as for pure Core-Based Routing introduced in Section 5.2. Next, the core $\boldsymbol{G}_{\text{core}} = (\boldsymbol{V}_{\text{core}}, \boldsymbol{E}_{\text{core}})$ is extracted from $\boldsymbol{G}_C$. Finally, we perform ALT preprocessing on $\boldsymbol{G}_{\text{core}}$ like presented in Section 5.1.2 without any modifications.

Landmarks and distances are computed on the lower-bound graph $\underline{\boldsymbol{G}_{\text{core}}}$. As worked out in Section 5.1.2, we use the MaxCover method when selecting up to 16 landmarks and Avoid if more then 16 landmarks shall be computed.

Node-Reordering. In order to improve spatial locality in our graph data structure, the nodes in $\boldsymbol{G}_C$ are reordered such that core nodes are at the beginning followed by component nodes. Please also refer to Appendix A.1.1 for a description of our graph data structure. As most of the query is performed on the core nodes, this improves cache efficiency yielding lower query times (see also [GKW07] and [Delo9a]). Furthermore, this allows us to use arrays of size $|\boldsymbol{V}_{\text{core}}|$ instead of $|\boldsymbol{V}_C|$ for storing the distances to/from each core node from/to each landmark, resulting in a significantly less amount of required space for the preprocessed data.

5.3.2. Query

The query algorithm for Core-ALT is basically the Core-Based Routing algorithm as described in Section 5.2.2. Phase one of the algorithm can be adopted without any modifications. The Dijkstra algorithm used during phase two on the core is exchanged by a multi-modal time-dependent ALT. However, the ALT algorithm requires distances from the landmarks to the target node t for computing a feasible potential function π. But the target node t might not be contained in the core. Thus, we require *proxy nodes* that assist by bounding the distance from v to t on the core.

Proxy Nodes. In [DSSW06] and [GKW07] the concept of proxy nodes has been introduced. Let $p \in \boldsymbol{V}_{\text{core}}$ be a core node with minimal distance $\text{dist}(t, p)$ from the target node t in $\boldsymbol{G}$. We call the node p *proxy node* for t. It can be virtually regarded as our 'target node' for the lower bound computations on the core regarding the triangle inequalities involving the landmarks. By the triangle inequality the following equation holds for the proxy and traget nodes

$$\text{dist}(v, p) \leq \text{dist}(v, t) + \text{dist}(t, p). \tag{5.18}$$

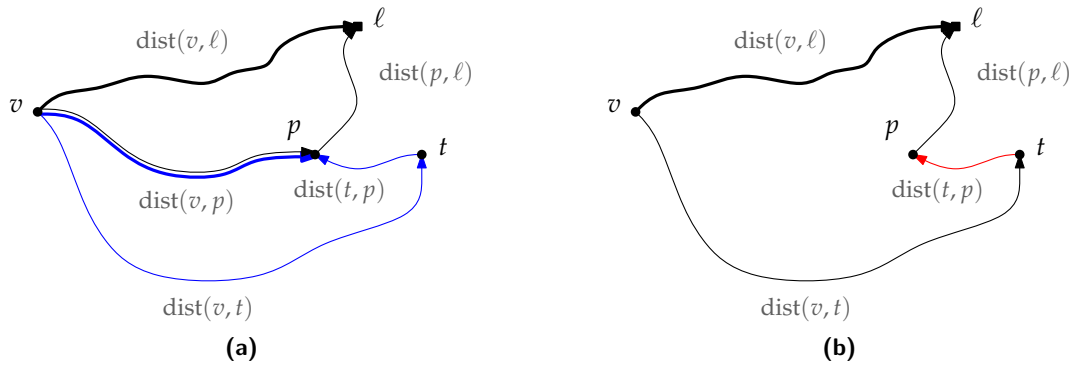


Figure 5.8.: Illustration of the triangle inequalities for core based ALT using proxy nodes p. In the left figure, the blue arrows show Equation 5.18 while the black arrows illustrate Equation 5.20 (the 'shortcut' is drawn bold in each case). When these two inequalities are combined, we obtain the right figure. For an s-t-query the distance of the red edge is unknown beforehand, thus, it has to be computed for each query separately.

Furthermore, for a landmark $\ell \in \boldsymbol{V}_{\text{core}}$ the following inequalities hold by the standard ALT approach.

$$\text{dist}(v,\ell) \leq \text{dist}(v,p) + \text{dist}(p,\ell) \quad \text{and} \tag{5.19}$$
$$\text{dist}(\ell,p) \leq \text{dist}(\ell,v) + \text{dist}(v,p). \tag{5.20}$$

Through substituting $\text{dist}(v,p)$ in Equations 5.19 and 5.20 by Equation 5.18, we derive the two inequalities

$$\text{dist}(v,\ell) \leq \text{dist}(v,t) + \text{dist}(t,p) + \text{dist}(p,\ell) \quad \text{and} \tag{5.21}$$
$$\text{dist}(\ell,p) \leq \text{dist}(\ell,v) + \text{dist}(v,t) + \text{dist}(t,p). \tag{5.22}$$

Since we are interested in a lower bound for $\text{dist}(v,t)$, we resolve these equations to $\text{dist}(v,t)$ yielding the feasible potential function

$$\begin{aligned}\pi(v) := \max\big\{ &\text{dist}(v,\ell) - \text{dist}(t,p) - \text{dist}(p,\ell), \\ &\text{dist}(\ell,p) - \text{dist}(\ell,v) - \text{dist}(t,p)\big\}.\end{aligned} \tag{5.23}$$

Note that if $\text{dist}(t,p)$ is given in advance, all remaining distance values are available on the core since all three nodes, v, p and the landmark ℓ belong to the core.

To obtain the distance $\text{dist}(t,p)$, we modify phase two of the core-based query algorithm as follows. In the beginning we grow a shortest path tree from t on the forward graph $\boldsymbol{G}$. As soon as we settle a node $v \in \boldsymbol{V}_{\text{core}}$ we stop and choose v as proxy node p. If the target is contained in the core itself, we stop immediately. However, even if the

target is not part of the core, the first core node is reached very quickly (cf. [Sch08a, p. 73]), making the runtime overhead of the additional preprocessing step required for each query almost negligible. Figure 5.8 illustrates the usage of landmarks in Equation 5.18 and 5.19.

The only further modification to the ALT algorithm is that we now require our lower bounds computed with regard to Equation 5.23. Apart from this, the multi-modal time-dependent ALT algorithm from Section 5.1.2 can be used directly.

5.3.3. Proof of Correctness

The formal proof of correctness for Core-ALT turns out relatively short, as we can simply relate to the correctness of Core-Based Routing and uni-directional ALT. Since Core-ALT makes use of Core-Based Routing the same restrictions that apply for Core-Based Routing also apply here. Hence, our proof is laid out for shortest path queries with respect to regular languages fulfilling the variable length property.

Theorem 4 (Correctness of Core-ALT)**.** *Multi-Modal Core-ALT is correct. Given a multi-modal graph $\boldsymbol{G}_C = (\boldsymbol{V}, \boldsymbol{E})$ after preprocessing $\boldsymbol{G}$, source and target nodes $s, t \in \boldsymbol{V}$, a departure time τ and a regular language L having the variable length property, applying the Multi-Modal Core-ALT algorithm on the contracted version $\boldsymbol{G}_C$ of $\boldsymbol{G}$ yields correct shortest paths from s to t at time τ wrt. the language L.*

Proof. By Theorem 3 the core-based query algorithm is correct. The only changes applied to the algorithm are the additional preprocessing step for determining the proxy node p of t and the replacement of the inner Dijkstra algorithm by uni-directional multi-modal ALT. The first has no influence on the correctness and the latter only influences paths that use the core. Hence, Core-ALT is correct for shortest paths not utilizing edges in the core.

In Section 5.3.2 we derived that the applied potential function through the proxy node is feasible. Hence, ALT restricted to the core is correct by the correctness of the ALT algorithm. This implies the correctness for shortest paths using the core by Theorem 3.

Altogether, we conclude that Core-ALT is correct. □

5.3.4. Discussion

In this section we presented Core-ALT. The main strengths of Core-ALT are, first of all, the very easy query algorithm. We can use the query algorithm for Core-Based Routing and only need to exchange the 'inner' part of it. The only significant modification is the computation of the proxy node p that is required for each s-t-query.

Second, preprocessing is straightforward, as we simply apply the standard ALT preprocessing algorithm to the lower bound core graph $\underline{G_{\text{core}}}$. Moreover, if the nodes of G_C are ordered such that all core nodes are at the beginning of our core data structure, we are able to reduce the required space to $\mathcal{O}(|\mathcal{L}| \cdot |V_{\text{core}}| + |\mathcal{L}|)$ as we can use arrays of length $|V_{\text{core}}|$ for storing landmark distances.

Hence, Core-ALT unifies the advantages of Core-Based Routing (the main part of the query is reduced to a small part of the original graph) with those of ALT (goal-directed searching) by eliminating the main disadvantage of ALT at the same time namely its high amount of preprocessed data, as we only need to compute landmark distances on the much smaller core graph.

We decided on using the uni-directional variant of the time-dependent ALT algorithm on the core. This is mainly due to the fact that bi-directional ALT requires a complicated query algorithm (see [NDLS08]). While in theory, bi-directional ALT could be augmented to multi-modality as well, it could be shown that query performance in fact drops when switching from uni-directional to bi-directional ALT in public transportation networks (see [Del09a]) due to the complicated query algorithm and bad lower bounds. Bi-directional ALT only improves over uni-directional ALT when shortest paths are approximated. As we believe that this decrease in query performance also applies to our multi-modal scenario, we decided against bi-directional ALT on the core.

In Section 6.4 we conduct experiments on various multi-modal networks using our Core-ALT algorithm which result in speed-ups up to 8.1.

5.4. Access-Node Routing

The previous speed-up techniques introduced in this work all had one goal in common. The query algorithm should work with automata as general as possible. Though, we already weakened this claim a little (our core-based techniques only allow automata that fulfill the variable length property), we now go one step further. In this section we introduce a speed-up technique called *Access-Node Routing* that yields very high speed-ups with the penalty of inflicting slightly more restrictions on the usage of valid automata during query.

Access-Node Routing is based on the assumption that the road network is only used at the beginning and the end of a journey. If this assumption holds, we observe that the number of 'relevant' entry points into the public transportation networks (rail and flight) is relatively small. Thus, in brief, the main idea of Access-Node Routing is to precompute for each node in the road network their 'relevant' entry points into the public transportation networks. These entry points are called *access-*

nodes. Furthermore, we also store their distances to and from each node in the road network. Hence, the query algorithm, when applied to nodes s and t in the road network, directly 'jumps' into the public transportation network through the access-nodes of s resp. t. This can be done extremely fast by table-lookups. The actual query is then restricted to the public transportation network, where we simply apply a multi-modal many-to-many query algorithm to route from the entry-access-nodes to the exit-access-nodes. If the shortest path does not use the public transportation network, we run an uni-modal time-independent shortest path query on the road network using one of today's speed-up techniques. This can be done in microseconds time [SS06b, BFM+07, Sch08a, Gei08, GSSD08, BD08, BD09] and does not become a burden. Altogether, with Access-Node Routing we are able to perform exact queries on intercontinental networks in a matter of milliseconds (see experimental Section 6.5).

Our Access-Node Routing technique adapts some of the ideas behind Transit-Node Routing (see [SS06b, BFM+07, BFSS07]). Transit-Node Routing is a time-independent speed-up techniques for road networks based on the observation that in road networks there are only a few number of important nodes (transit nodes), where all 'far' shortest paths go through. Thus, the basic idea is to compute distance tables for each road in the road network to their relevant transit nodes. Furthermore, a distance table is computed between all pairs of transit nodes (which can be done since the number of transit nodes is low). A query algorithm then determines the relevant transit nodes for the source s and target t, looks up the distances between them in the distance table and selects the minimum 'path' from all combinations. In order to determine whether the shortest path does not use transit nodes, a locality filter is used. These local queries are computed using a classic shortest path algorithm without using the transit nodes.

This section is made up as follows. First, we motivate our Access-Node Routing approach by showing the restrictions concerning automata and arguing that there are use cases where these restrictions are not harmful. We then give a formal introduction explaining the technique in more detail, especially the term access-node. After that, we examine preprocessing, where we describe how we compute relevant access-nodes in detail. It turns out that computing exact access-nodes involves profile searches on the whole graph. This is unfeasible as preprocessing times are too high. Thus, we present a fast preprocessing technique that computes approximate access-nodes, i.e., there might be too many access-nodes obtained for a given road network node.

Next, the query algorithm is presented which is basically a multi-modal time-dependent Dijkstra on the public transportation network where the source and target nodes S and T are the access-nodes of s and t, respectively. For very large networks storing access-nodes for *every* node of the road network is not feasible, as the required space consumption is too high. Thus, we show how Access-Node Routing can be com-

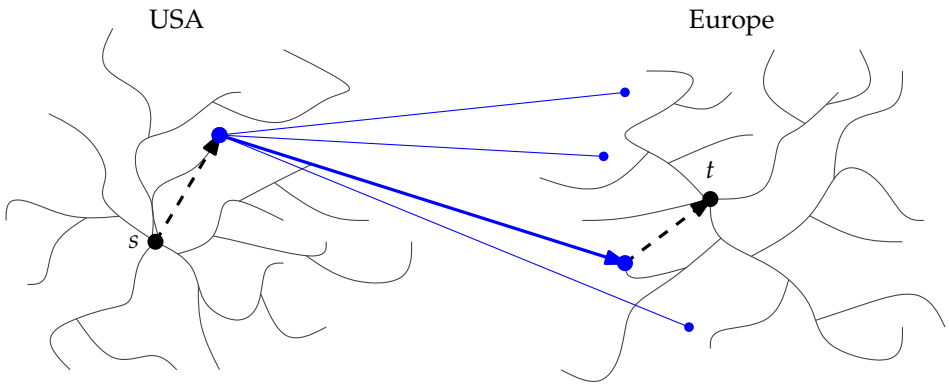


Figure 5.9.: An intercontinental query from USA to Europe. With a 'classic' (bi-directional) routing approach big parts of the road network in Europe and the US would be explored (black). Access-Node Routing only looks at the flights (blue). The distances to the relevant airports in the road network from s and t are precomputed.

bined with our Core-Based Routing approach from Section 5.2 yielding a technique we call *Core-Based Access-Node Routing*. By these means access-nodes have to be computed on the much smaller core graph alone. While the space consumption can be reduced radically, the slowdown in query performance is negligible (see Section 6.5). Finally, we provide a proof of correctness for Access-Node Routing, winding up this section with a small discussion.

5.4.1. Motivation

When planning a far voyage (e.g., across Europe, or even from Europe into the United States), it is reasonable to assume that we do not want to use our car the whole trip. Instead, what we like to do is to use means of public transportation for the most part of the journey. For example, we use our car to get to the airport, take a flight that covers most of the distance and at the end reach our final destination by train. The only parts where we might not be able to use means of public transportation (and therefore require the road network) are the first part and the last part of the journey.

Although there are instances where we are required to use the road network in between of the voyage (for example if we transfered in Paris from Gare du Nord to Gare de l'Est), this can be generally assumed as undesirable. Actually, we expect that the public transportation network is gapless, i.e., we can travel between two points in the public transportation network without reverting to the road network (car or foot). Thus, it is reasonable to assume that the road network shall be used only at the beginning and the end of the journey in order to reach resp. leave the public transportation network.

Access-Node Routing is tailored for these applications, as we 'skip' the road network and reduce the task of finding a route to the public transportation network. A main application is intercontinental queries. If we want to travel from America to Europe, we *have* to use the flight network, as there are no other connections between these two continents in our graphs. Furthermore, the flight network covers very far distances by only very few edges. A standard DIJKSTRA search involving the road network would therefore settle all nodes with smaller distance, which results in a very large search space of unnecessarily explored nodes. Our access-node based approach does not settle any nodes in the road network, thus, the query is reduced to the very small flight network, which results in very small query times. Figure 5.9 illustrates this issue.

5.4.2. Formal Introduction

In this section we describe Access-Node Routing more formally. Especially, we develop some notion and give a definition of the term *access-node*. In the following, we are given multi-modal graphs $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ with labeled edges. The graphs consists of both a road network and at least one public transportation network.

The main idea of the speed-up technique is to access relevant nodes of the public transportation network directly from the road network in order to avoid the expensive search in the road network. However, not every node in the public transportation network is suited as 'access-node'. For example, in the flight network the departure and arrival nodes are used to model internal procedures at airports and should not be accessed directly from the road network. Therefore, we give a formal definition which nodes of the public transportation network are *access-node candidates*.

Definition 9 (Access-Node Candidate)**.** *Let $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ be a multi-modal graph. A node $v \in \boldsymbol{V}$ is called* access-node candidate *if the following properties hold.*

(i) The node v is not part of the road network, thus, $\text{label}(v) \neq$ `ROAD_NODE`*,*

(ii) The node v is linked to the road network, i.e., there is a neighbor w of v for which $\text{label}(w) =$ `ROAD_NODE` *holds.*

The set of all access-node candidates is denoted by A.

Note that the condition (ii) implies `LINK_EDGE` labeled edges between v and w and vice versa. According to the linking rules (cf. Section 3.5), in our case access-node candidates of the railway network are exactly the station nodes and access-node candidates of the flight network are the airport nodes. These are the respective nodes linked to the road network. Nodes $v \in A$ can be interpreted as entry (or exit) points

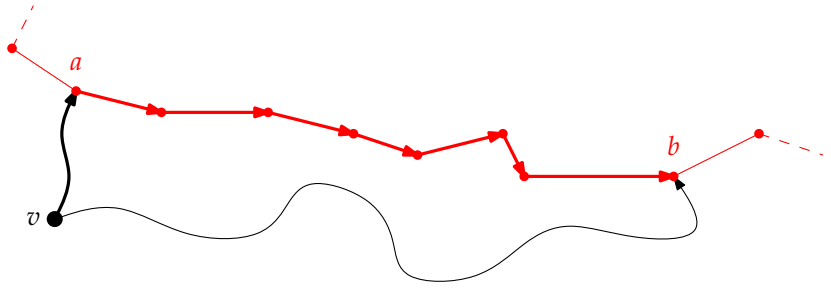


Figure 5.10.: Illustration of Definition 10 (access-nodes). The node a in the railway network (red) is an access-node for the road network (black) node v, as the shortest path (bold) to the access-node candidate b uses a to enter the public transportation network (for some departure time τ at v). The black path using solely the road network is longer.

to/from the public transportation network. Computing distances from every road network node to every access-node would require too much space. Moreover, not every entry point to the public transportation network is mandatory for correct shortest paths. Thus, we now give an exact definition of a node being an *access-node* for some road network node v.

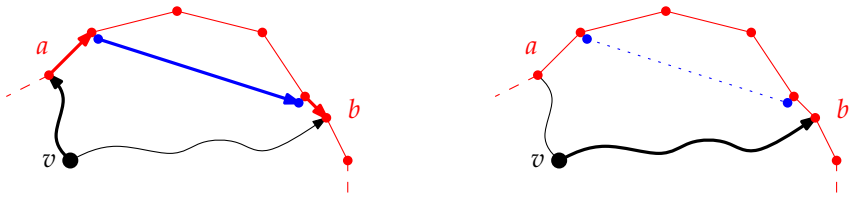
Definition 10 (Access-Node)**.** *Let $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ be a multi modal graph with access-node candidate set A. Furthermore, let $v \in \boldsymbol{V}$ be a node belonging to the road network, i.e.,* $\text{label}(v) = \texttt{ROAD\_NODE}$.

Then a node $a \in A$ is called access-node *for v, if there exists another access-node candidate $b \in A$ and a departure time τ for which the shortest path from v to b at time τ uses the node a to enter the public transportation network. Once entered, the shortest path is restricted to solely using edges of the transportation network.*

The set of access-nodes for a certain road node v is denoted by $A(v)$.

Please also refer to Figure 5.10 for a visualization of this definition. When we used the term 'shortest path' in our definition, we deliberately omitted to specify the form of shortest path in more detail. The only restriction mentioned is that the shortest path may not leave the public transportation network once entered. Apart from that we are free to use any regular language L according to which the shortest paths should conform to. However, the set of access-nodes depends on the language L. For that reason, we fix the automaton during access-node computation. See also Figure 5.11 for an illustration.

The set $A(v)$ of access-nodes belonging to a certain road network node v can be imagined as those entry points to the public transportation network that are at least once relevant over the day in the sense that for that specific time it pays off going through one of the access-nodes to reach one point in the public transportation net-



(a) Allowing Rail and Flight. **(b)** Only allowing Rail.

Figure 5.11.: Comparison on the effect of access-nodes when different automata are used during preprocessing. To the left, a is an access-node for v since the shortest path (black) uses the very fast flight network (blue) to reach b. On the right, b is reached through the road network (black) the most quickly (for every τ). Thus, a might not be an access-node for v.

work.

If for a real shortest s-t-path query, we enter the public transportation network through an access-node a, we eventually need to leave the public transportation network if the target is contained in the road network. For that reason, we need to compute *backward access-nodes* that are the relevant exit points from the public transportation network. A backward access-node for v is basically an access-node of v in the backward graph $\overleftarrow{G}$.

Definition 11 (Backward-Access-Node). *Let $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ be a multi modal graph with access-node candidate set A. Furthermore, let $v \in \boldsymbol{V}$ be a node belonging to the road network, i.e.,* $\text{label}(v) = \texttt{ROAD\_NODE}$.

Then $\overleftarrow{A}(v) \subset \boldsymbol{V}$ denotes the set of backward access-nodes. *A node a is contained in $\overleftarrow{A}(v)$ if a is an access-node for v in the backward graph $\overleftarrow{G}$.*

So, if for some nodes $s, t \in \boldsymbol{V}$ a shortest s-t-path through the public transportation network at some time τ should be computed, we know the points that are relevant for entering the public transportation network by the forward access-nodes $A(s)$. The relevant points for exiting the public transportation network are given by $\overleftarrow{A}(t)$. Thus, we can restrict ourselves to finding a shortest path from the set $A(s)$ to $\overleftarrow{A}(t)$ in the public transportation network by any multi-modal shortest path algorithm. The final path is then combined from the distances of the respective access-nodes plus the length of the path in the public transportation network.

5.4.3. Preprocessing

Preprocessing of Access-Node Routing involves the computation of the sets $A(v)$ and $\overleftarrow{A}(v)$ for all $v \in \boldsymbol{V}$ with $\text{label}(v) = \texttt{ROAD\_NODE}$. As we need to check whether an

access-node candidate is relevant at least once over the day, we need to compute profile searches at some point during preprocessing. In this section we present two alternatives for computing the access-nodes. The first alternative is a more or less direct application of Definition 10 and involves the computation of the access-node sets $A(v)$ directly by running a one-to-all profile search from v collecting the relevant access-nodes along the way.

The second method computes the inverse relation A^{-1}, i.e., for each access-node candidate $a \in A$ we obtain those nodes v in the road network for which a is an access-node. Moreover, we allow approximations of the sets A^{-1} in the sense that they may contain too many nodes. While this leads to slightly more access-nodes per road node, it only requires us to compute a backward profile search restricted to the public transportation network. Preprocessing times are thus reduced drastically by this approach, as the public transportation network is only a small fraction of the size of the whole multi-modal network $\boldsymbol{G}$.

We describe our preprocessing algorithms only using the computation of the forward access-nodes sets $A(v)$. The backward access-nodes are computed analogously by switching $\boldsymbol{G}$ with $\overleftarrow{\boldsymbol{G}}$.

Alternative I: Exact Access-Nodes. Regarding the first alternative for computing access-nodes, the preprocessing algorithm consists of a time-dependent multi-modal profile search algorithm executed for each node $v \in \boldsymbol{V}$ that is a member of the road network. Refer to Section 4.2.2 for an introduction to profile queries. Recall that our piecewise linear functions only support homogeneous gradients. Thus, we cannot use the 'simple' label correcting algorithm from [Dea99] but have to evade to the multi label correcting variant as shown in Algorithm 4 on page 50. Additionally, for each label we introduce a flag, covered that denotes whether a path toward the resp. label has already entered the public transportation network once.

For each road node v the algorithm is then executed as follows. In the beginning we set $A(v) = \emptyset$. Moreover, we use an automaton that does not allow the usage of the road network between two subsequent parts in the public transportation network (e.g., a hierarchical automaton). Furthermore, we insert all source labels with the covered flag set to `false`. The covered flag is then propagated through the graph, i.e., during edge relaxation of an edge $e = (v, w)$ its function is linked into the current label (cf. Algorithm 4, line 6) with the covered flag of f_v being passed to the new function.

Then, each time we are about to enter the public transportation network, i.e., we insert a node $a \in A$ with label f_a^{new}, we check f_a^{new} against all labels already assigned to a. If there is at least one label f_a for which $f_a < f_a^{\text{new}}$ does not hold, we set covered$(f_a^{\text{new}}) =$ `true` and insert a into the set $A(v)$ of access-nodes for v.

If on the other hand, we use an edge in the public transportation network toward an access-node, i.e., we insert a node $a \in A(v)$ whereas the current label already has its covered flag set, we can remove a from $A(v)$ if the following holds. For all labels f_a assigned at a we determine those labels for which the inequality $f_a < f_a^{\text{new}}$ does not hold. If of all of these labels their predecessors have their covered flag set, i.e., all paths toward a go through the public transportation network, a can be removed from $A(v)$, as the shortest path to a is now reached through the public transportation network during all times of day.

The procedure stops as soon as the priority queue is empty, or it holds that all remaining labels in the priority queue have their covered flag set. In the latter case, every path contained in the search space entered the public transportation network already at some point. Thus, every undiscovered access-node candidate can be reached through the public transportation network.

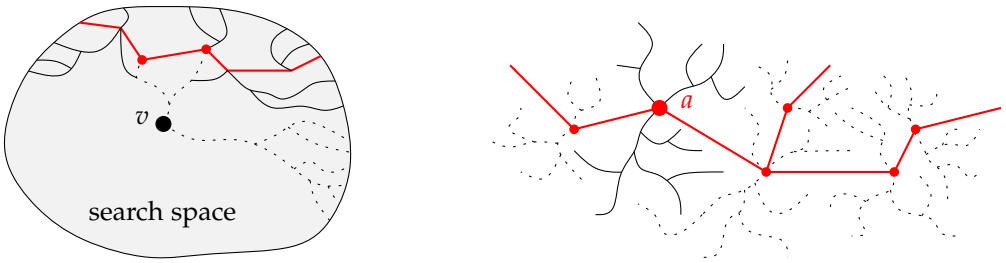
Issues. While this alternative of computing access-nodes seems straightforward as it implements Definition 10 more or less directly, it has some major drawbacks. First of all, we need to compute one-to-all profile queries on $\boldsymbol{G}$. As our largest graphs have about 50 million nodes, computing a one-to-all profile query takes much too long. Moreover, the largest parts of the multi-modal networks are made up of the road network. This implies two aspects. First, there are a lot of nodes $v \in \boldsymbol{V}$ for which access-nodes have to be computed (every computation yielding in a full one-to-all profile query). Second, the fact that the largest portions of the graph are time-independent is not accounted for as the profile search is conducted on these edges as well.

Altogether, an early experimental test of this algorithm, exposed it as unpractical as the computation simply took too long (one profile query taking several hours). Hence, we present an alternative which remedies some of the above issues.

Alternative II: Approximating Access-Nodes. Instead of computing for every road note $v \in \boldsymbol{V}$ the set of access-nodes $A(v)$, the idea of the second alternative is to compute for every access-node candidate $a \in A$ the inverse relation $A^{-1}(a)$. Thereby, A^{-1} is defined as follows.

$$\forall v \in \boldsymbol{V}, \forall a \in A: \quad v \in A^{-1}(a) :\Leftrightarrow a \in A(v). \tag{5.24}$$

Roughly speaking, the set $A^{-1}(a)$ for a given access-node candidate a contains all road nodes $v \in \boldsymbol{V}$ for which a is an access-node for them. Thus, our approach is to compute the relation $A^{-1}(a)$ for all access-node candidates and invert $A^{-1}(a)$ to obtain $A(v)$ afterward.



(a) Alternative I. **(b)** Alternative II.

Figure 5.12.: Visualization of both Alternatives to compute access-nodes.
Figure (a) illustrates the search space of the multi label correcting algorithm. Uncovered paths are dotted. The red edges belong to the railway network. The red nodes belong to $A(v)$, as they are reached by uncovered paths. Moreover, the search cannot be aborted yet, as there are uncovered paths active in the queue (at the border of the search space).
Figure (b) illustrates computation of approximate access-nodes. From all access-node candidates in the transportation network backward searches are conducted in the road network with upper bounds toward a as initial keys. The road nodes in the undotted search space may use a as access-node, as for the other nodes it always pays off to reach a through the public transportation network.

This approach is not exact in the sense that it computes the minimal relation A^{-1} that is required for correct shortest path queries. Instead, the set $A^{-1}(a)$ might contain too many nodes. However, this is not a problem to correctness of our query algorithm (see later).

Our algorithm works in two phases. Given an access-node candidate a, we first compute a full multi-modal backward profile search on the public transportation network of $\overleftarrow{G}$ originating from a. This results in travel time functions f_b at each access-node candidate $b \in A$ that represents the time to get from b to a in the public transportation network for any given time of day.

Now, what we are are looking for are all road nodes $v \in \boldsymbol{V}$ for which $a \in A(v)$ holds, i.e., there is at least one time of day during which another access-node candidate $b \in A$ is reached by entering the public transportation network through a. These road nodes are then contained in $A^{-1}(a)$. Thus, the second phase consists of a uni-modal time-independent many-to-all backward Dijkstra search restricted to the road network. The priority queue is initialized with all access-node candidates $b \in A$ and their upper bounds $\overline{f}_b$ as keys. Furthermore, the access-node a is inserted with key 0 and a special covered flag is set to `true`. Again, the covered flag is propagated through the backward graph $\overleftarrow{G}$. Finally, each time a node $v \in \boldsymbol{V}$ is settled, we insert v into $A^{-1}(a)$ if and only if covered(v) evaluates to `true`.

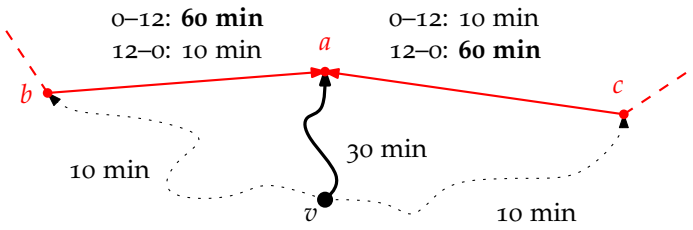


Figure 5.13.: Approximating access-nodes. The node v is contained in $A^{-1}(a)$ since when using upper bounds at b and c, both indirect paths through b resp. c are longer than the direct path to a. In fact, for every time of day one of the indirect paths is shorter: The v-c-a-path in the morning and the v-b-a-path in the afternoon. Hence, a would not required to be an access-node for v.

This can be interpreted as follows (cf. Figure 5.12b). For nodes v that do not have their covered flag enabled, it *always* pays off using a node other than a to enter the public transportation network since the shortest v-a-path always goes through another access-node candidate b (we use the upper bounds for f_b to account for the 'worst' connection during the day). On the other hand, for those nodes v having their covered flag enabled, it pays off using the road network to get to a, thus, a might be an access-node. Note that a does not necessarily has to be an access-node for every $v \in \boldsymbol{V}$ with $\text{covered}(v) = \texttt{true}$ since using upper bounds instead of exact profile functions might turn out too conservative. See also Figure 5.13 for an example on this issue.

Performance. The main drawback of the exact approach is performance as worked out above. This is due to the fact that we are required to perform full profile searches on the entire graph. Our approximate approach, however, only requires (backward) profile searches from each access-node candidate $a \in A$. Moreover, these profile searches are restricted to the public transportation network, which make up only small parts of our graphs. However, with an increasing size of the public transportation network the performance degrades badly as well, as the profile search becomes the main bottleneck during preprocessing.

Automata. When describing the preprocessing algorithms, we omitted to talk about automata or multi-modality. In fact, this topic requires some attention. Regarding exact access-nodes, we have to use an automaton that only allows the road network in the beginning or the end of a shortest path. This is covered by the following definition.

Definition 12 (Enclose Property)**.** *Let L be a regular language over the alphabet Σ of edge-*

labels. If L is of the form

$$L = \underbrace{\sigma_{r_1}^*}_{L_{r_1}} \; l \; \underbrace{\sigma_t^*}_{L_t} \; l \; \underbrace{\sigma_{r_2}^*}_{L_{r_2}}, \tag{5.25}$$

where $\sigma_{r_1}, \sigma_{r_2} \in \{\texttt{CAR\_EDGE}, \texttt{FOOT\_EDGE}\}$ *and* $\sigma_t \in \Sigma \setminus \{\texttt{CAR\_EDGE}, \texttt{FOOT\_EDGE}\}$ *and* $l = \texttt{LINK\_EDGE}$*, we say that L has the* enclose *property (the public-transportation part of L is enclosed by the road network part).*

Regarding approximate access-nodes, we only use a multi-modal query on the public transportation network for computing the backward profile search. Thus, to compute access-nodes wrt. a language L (having the enclose property), we use the (inverse) automaton representing only the L_t part of L. The rest is incorporated into the uni-modal search on the road network as follows. When computing forward access-nodes, we restrict ourselves to the foot/car edges in the road network depending on the value of σ_{r_1}. Analogously, for computing backward access-nodes, we restrict the road network query to the foot/car edges depending on the value of σ_{r_2}.

5.4.4. Query

In this section we present our query algorithm for Access-Node Routing. In the following we are given a multi-modal graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ with Σ-labeled edges and sets of forward access-nodes $A(v)$ and backward access-nodes $\overleftarrow{A}(v)$, as well as source and target nodes $s, t \in \boldsymbol{V}$ and a departure time τ. Distances to and from all access-nodes $a \in A(v)$ resp. $a \in \overleftarrow{A}(v)$ from and to every road network node v are available. Moreover, we are given a finite automaton $\mathcal{A} = (\Sigma, Q, \delta, S, F)$ (also used for preprocessing) that represents a regular language L having the enclose property.

Note, if either s or t are not part of the road network, we set $A(s) := \{s\}$ and $\overleftarrow{A}(t) := \{t\}$, respectively. Now we basically use an uni-directional time-dependent multi-modal query algorithm to compute shortest paths from $A(s)$ to $\overleftarrow{A}(t)$. This can be done by Algorithm 5 with a few minor modifications.

Query between Access-Nodes. Let Q_{entry} denote the set of states that can be reached through a link into another network from any initial state of the automaton, i.e.,

$$Q_{\text{entry}} := \bigcup_{q_s \in S} \delta(q_s, \texttt{LINK\_EDGE}). \tag{5.26}$$

Note that we defined access-node candidates as nodes in the public transportation network which have an incident LINK_EDGE labeled edge. Moreover, the form of the

language L implies that the automaton supports transitions into the public transportation network by LINK_EDGE emerging from the initial states. Thus, the priority queue of the multi-modal shortest path algorithm is initialized by all product-nodes (a, q) where $a \in A(s)$ and $q \in Q_{\text{entry}}$. The key of (a, q) is set to $\text{dist}(s, a)$ which is determined by a table-lookup from the precomputed data.

Analogously, let Q_{exit} denote the set of states in Q that have a LINK_EDGE labeled transition to a final state, i.e.,

$$Q_{\text{exit}} := \{q \in Q \mid \exists q_f \in F \text{ and } q_f \in \delta(q, \texttt{LINK\_EDGE})\}. \tag{5.27}$$

Then the target set T for the query algorithm contains all product-nodes (a, q) where $a \in \overleftarrow{A}(t)$ and $q \in Q_{\text{exit}}$.

The multi-modal query is restricted to the public-transportation network (which is enforced by the automaton anyway) and can be stopped as soon as all product-nodes from the target node set T have been settled. Note that the edge weights along the time-dependent edges in the public transportation network have to be evaluated with respect to τ.

Selecting the Path. After the query algorithm finished, we obtain for each product-node $(a, q) \in T$ a distance label $\text{dist}_s\big((a, q)\big)$ from the source node s (since we initialized the priority queue with the correct distances from the source node s). Furthermore, the distance to the target node t from a product-node (a, q) can be obtained by $\text{dist}(a, t)$ through a table-lookup. The final product-node (a, q) inducing the shortest s-t-path is, thus, obtained by selecting (a, q) according to

$$(a, q) := \underset{(a,q) \in T}{\text{argmin}} \Big\{ \underbrace{\text{dist}_s\big((a, q)\big) + \text{dist}(a, t)}_{\text{dist}(s,t)} \Big\}. \tag{5.28}$$

Local Paths. While the algorithm described above yields correct shortest s-t-paths if they truly require the public transportation network, i.e., a standard multi-modal algorithm would also use the public transportation network, shortest paths using solely the road network are not found. More precisely, our Access-Node Routing approach *forces* the usage of the public transportation network, thus, an incorrect (i.e., not shortest) s-t-path through the public transportation network is computed. See Figure 5.14 for an example. The same problem arises with Transit Node Routing (see [SS06b]). The solution is to introduce a *locality filter* $L : \boldsymbol{V} \times \boldsymbol{V} \to \{\texttt{true}, \texttt{false}\}$, which decides for two arbitrary nodes $s, t \in \boldsymbol{V}$ if a *local search* solely on the road network is required. In our case, this local search can be performed by any time-independent uni-modal shortest path algorithm.

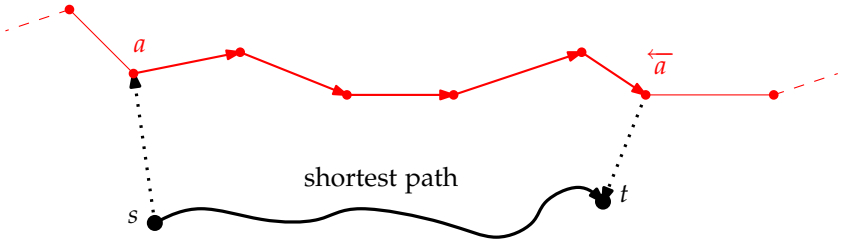


Figure 5.14.: If the source and target nodes are too close to each other, the shortest path (bold) may not use the public transportation network. In this case, using the access-nodes (dashed lines) would yield a false result. Thus, a separate time-independent uni-modal shortest path computation needs to be computed restricted to the road network.

Note that $L(s,t) = \texttt{true}$ should always imply a local search. However, although $L(s,t) = \texttt{false}$ does not require a local search, computing the local search additionally does not harm correctness of the algorithm if we select the path of minimum length between the local and the public transportation path in the end. Today's time-independent speed-up techniques for road networks are fast enough to handle queries in continental-sized graphs in a matter of microseconds [Delo9a]. This effort is almost negligible, thus, we are able to use a very conservative locality filter defined by

$$\forall s,t \in \boldsymbol{V}: \quad L(s,t) = \texttt{true} \Leftrightarrow s \text{ and } t \text{ are in different strongly connected components in the road network.} \tag{5.29}$$

This always induces a local search in the road network (even for far queries, e.g., across Europe) if there exists a path from s to t in the road network. Only for intercontinental queries we omit the search.

5.4.5. Core-Based Access-Node Routing

One of the main drawbacks of pure Access-Node Routing is the high space consumption concerning the precomputed data. For each node v of the road network we have to store sets $A(v)$ of forward and $\overleftarrow{A}(v)$ of backward access-nodes. Moreover, for each of the forward resp. backward access-nodes we have to store the distances to/from the access-nodes. Although the number of access-nodes per road node is relatively small, the space consumption is quite high for our largest networks as the number of road nodes is high.

For that reason, Access-Node Routing can be combined with Core-Based Routing (cf. Section 5.2) in a straightforward manner. The resulting speed-up technique is called *Core-Based Access-Node Routing* and remedies the disadvantage of the high space consumption by computing access-nodes only on the much smaller core graph $\boldsymbol{G}_{\text{core}}$.

Preprocessing. Preprocessing for Core-Based Access-Node Routing is done in two steps. First, we contract the graph $\boldsymbol{G}$ to $\boldsymbol{G}_C$ like described in Section 5.2.1 and extract the core yielding a graph $\boldsymbol{G}_{\text{core}} = (\boldsymbol{V}_{\text{core}}, \boldsymbol{E}_{\text{core}})$. In a second step we apply one of the preprocessing routines for Access-Node Routing as presented in Section 5.4.3. Although the core graph is much smaller, in our experiments we continue using the approximate approach for computing access-nodes, as preprocessing times for computing exact access-nodes are still too high.

Node-Reordering. Similar to Core-ALT (cf. Section 5.3.1), we reorder the nodes of $\boldsymbol{G}_C$ such that nodes belonging to $\boldsymbol{G}_{\text{core}}$ are in front of the component nodes. Again, this has the effect of improving spatial locality and, thus, yielding higher cache-hit-rates which improves query performance. Moreover, with regard to our static graph data structure (cf. Appendix A.1.1), it allows us to use smaller arrays for storing the access-nodes.

Query. The query algorithm is basically the core-based query algorithm introduced in Section 5.2.2 where the 'inner' algorithm during phase two is exchanged by our access-node query algorithm from Section 5.4.4. Phase one of the core-based query algorithm is performed without any modification. Assume that the shortest path is not completely contained in the component and we are given a set S of core-entry-product-nodes as well as a set T of core-exit-product-nodes. Then phase two (which is now our access-node based approach) must find shortest paths from all nodes S to all nodes T. This requires some extra consideration.

The Access-Node Routing algorithm is modified as follows. Basically, we need to consider all access-nodes for every core-entry-product-node from S as well as every backward access-node for every core-exit-product-node from T. However, the same node $a \in A$ may be an access-node for two different core-entry-product-nodes $(v_1, q_1), (v_2, q_2) \in S$. Thus, we insert all product-nodes (a, q') into the forward queue where there exists at least one product-node $(v, q) \in S$ with $a \in A(v)$ and $q' \in \delta(q, \texttt{LINK\_EDGE})$. The key of (a, q') is set to

$$\text{key}(a, q') := \min \Big\{ \text{dist}_s\big((v, q)\big) + \text{dist}\big((v, q), (a, q')\big) \mid (v, q) \in S \text{ and } a \in A(v) \text{ and } q' \in \delta(q, \texttt{LINK\_EDGE}) \Big\}. \quad (5.30)$$

In other words, we obtain the key by taking the minimum length of all paths from the source s over a core-entry-product-node (v, q) toward one of its access-nodes (a, q').

The same is true for the backward-access-product-nodes that are used as target nodes for the multi-modal search on the public transportation network. Hence, the set

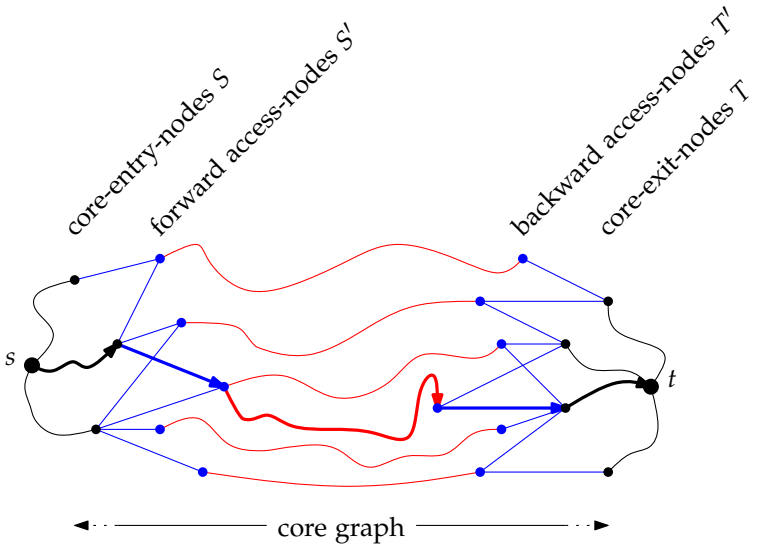


Figure 5.15.: Visualization of a query of the Core-Based Access-Node Routing algorithm. The (bi-directional) component query is drawn in black, the shortcuts to the access-nodes are blue and the shortest paths between the forward and backward access-nodes in the core are drawn in red.

T' of backward access-product-nodes contains all product-nodes (a, q') where there is at least one product-node $(v, q) \in T$ with $a \in \overleftarrow{A}(v)$ and $q' \in \delta^{-1}(q, \texttt{LINK\_EDGE})$.

After the multi-modal query algorithm on the public transportation network has finished (i.e., the priority queue runs empty or all backward access-product-nodes from T' have been settled), we need to set the distance labels at the core-exit-nodes properly. For every core-exit-product-node (v, q) we have a distance label $\overleftarrow{\text{dist}}_t((v, q))$ to its respective target product-node in the component. This label is computed in phase one of the core-based query algorithm by the time-independent backward-search on the component. Thus, for every core-exit-product-node $(v, q) \in T$ we set its predecessor along the shortest path to the backward access-product-node (a, q') defined by

$$(a, q') := \underset{\substack{(a,q') \in T', \\ a \in \overleftarrow{A}(v) \text{ and} \\ q' \in \delta^{-1}(q, \texttt{LINK})}}{\text{argmin}} \left\{ \text{dist}_s((a, q')) + \text{dist}((a, q'), (v, q)) + \overleftarrow{\text{dist}}_t((v, q)) \right\}. \quad (5.31)$$

By these means we ensure correct forward and backward distance labels at the core-exit-product-nodes, thus, the path selection routine from the Core-Based Routing approach can be adopted without modifications (See the paragraph about combining the shortest path in Section 5.2.2 on page 87). Please also refer to Figure 5.15 which illustrates the steps behind the Core-Based Access-Node Routing algorithm.

5.4.6. Proof of Correctness

In this section we give formal proofs of correctness. At first, we prove that the basic Access-Node Routing algorithm is correct. On top of that, we show that Core-Based Access-Node Routing is also correct.

We first show through a series of lemmas that our preprocessing routine discovers all forward and backward access-nodes for every node in the road network. Moreover, we show that every shortest s-t-path that does not solely use the road network has to go through the forward and backward access-nodes $A(s)$ resp. $\overleftarrow{A}(t)$. With these assumption proven, we then show the correctness of our query algorithms.

In the following we are given a multi-modal and Σ-labeled graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ that contains both a road network and at least one public transportation network. Furthermore, we are given a finite automaton $\mathcal{A} = (\Sigma, Q, \delta, S, F)$ that represents a regular language L that conforms to the enclose property (see Definition 12).

Lemma 3 (Exact Access-Node Preprocessing)**.** *For each node $v \in \boldsymbol{G}$ where* label$(v) =$ *ROAD_NODE, the algorithm for exact access-node preprocessing (cf. Section 5.4.3 on page 101) discovers all access-nodes conforming to Definition 10.*

Proof. We prove this lemma by contradiction. Assume there is a node $a \in A(v)$ that is not discovered by the preprocessing algorithm. Then, according to Defintion 10 there exists an access-node-candidate $b \in A$, $a \neq b$ and a departure time τ such that the shortest v-b-path P at departure time τ at v uses a to enter the public transportation network, i.e.,

$$P = \big[(v, q_1), \ldots, (a, q_i), \ldots, (b, q_k)\big], \tag{5.32}$$

where $q_i \in Q$.

According to the preprocessing algorithm, each time we insert an access-node candidate $a \in A$ into the priority queue by a label that is still uncovered, we insert the node a into the set $V(a)$. In P for all product-nodes (v_j, q_j) before (a, q_i) it holds that label$(v_j) =$ `ROAD_NODE`, thus, the labels in our preprocessing algorithm along this subpath are all uncovered. Consequently, either one of the following cases must hold.

1. The subpath $P' = \big[(v, q_1), \ldots, (a, q_i)\big]$ in the road network is not found by our preprocessing algorithm. This is a contradiction to the correctness of the multi label correcting algorithm.

2. The node a was removed from $A(v)$ at a later time during the run of the algorithm. In that case, we insert the node a into the priority queue by a label $f_a^{\text{new}} = f^{\text{current}} \oplus f_e$ through an edge $e \in \boldsymbol{E}$ with covered$(f^{\text{current}}) =$ `true`, i.e.,

we are coming through the transportation network toward a. Moreover, for all labels f_a assigned to a it has to hold that $f_a < f_a^{\text{new}}$ and $\text{covered}(f_a) = \texttt{true}$. As a consequence, every shortest v-a-path for every departure time τ is reaching a through the transportation network. This is a contradiction to P' being a shortest path.

Altogether, we conclude that every access-node $a \in A(v)$ is found by our preprocessing algorithm for exact access-nodes. □

Lemma 4 (Approximate Access-Node Preprocessing)**.** *For each node $v \in \boldsymbol{G}$ having* $\text{label}(v) = \texttt{ROAD\_NODE}$*, the algorithm for approximate access-node preprocessing (cf. Section 5.4.3 on page 102) discovers all access-nodes conforming to Definition 10.*

Proof. Again, we prove this lemma by contradiction. Let $a \in A(v)$ be an access-node of v that is not discovered by the preprocessing algorithm. There must exist an access-node candidate $b \in A$, $a \neq b$ and a departure time τ where the shortest v-b-path P at time τ uses a to enter the public transportation network, i.e., $P = \big[(v, q_1), \ldots, (a, q_i), \ldots, (b, q_k)\big]$, where $q_i \in Q$ and for all product-nodes (v_j, q_j) that are in front of (a, q_i) it holds that $\text{label}(v_j) = \texttt{ROAD\_NODE}$.

Let $P' = \big[(v, q_1), \ldots, (a, q_i)\big]$ denote the subpath of P from v to a. Because every subpath of a shortest path is again a shortest path, P' has to be a shortest v-a-path for the departure time τ. Our approximate preprocessing algorithm first runs a backward profile search on the public transportation network originating at a. Then, from every access-node candidate from A we start a time-independent backward search on the road network simultaneously. Moreover, the distances of every node from A are initialized with the upper bound of the travel time function from b to a. In order for $a \neq A(v)$ (according to the algorithm), the node v is on the shortest path tree originating from one of the access-node candidates $b \in A$, $b \neq a$. Thus, by the preprocessing algorithm it holds that

$$\text{dist}(v, b) + \overline{\text{dist}^*}(b, a) < \text{dist}(v, a). \tag{5.33}$$

Because this is true for the upper bound of $\text{dist}^*(b, a)$, this implies

$$\text{dist}(v, b, \tau) + \text{dist}(b, a, \tau + \text{dist}(v, b, \tau)) < \text{dist}(v, a, \tau) \quad \forall \tau < \Pi. \tag{5.34}$$

Thus, the shortest path induced by the preprocessing algorithm through b toward a is shorter for all departure times τ than the path P'. This is a contradiction to P' being a shortest path and we conclude that every access-node $a \in A(v)$ is also found by our approximate preprocessing algorithm. □

We have proven that both preprocessing algorithms presented are correct wrt. to Definition 10 of the term access-node. For the correctness of the whole Access-Node Routing approach, we now turn toward proving that every shortest s-t-path in $\boldsymbol{G}$ is also found by the Access-Node Routing query algorithm.

Lemma 5. *Let P be a shortest s-t-path in $\boldsymbol{G}$ for some departure time τ at s conforming to a language L having the enclose property. Furthermore, the shortest path is not using the road network alone. If $a \in \boldsymbol{V}$ is the first node in P with* $\text{label}(a) \neq$ `ROAD_NODE` *and $\overleftarrow{a} \in \boldsymbol{V}$ the last node in P with* $\text{label}(\overleftarrow{a}) \neq$ `ROAD_NODE`*, then $a \in A(s)$ and $\overleftarrow{a} \in \overleftarrow{A}(t)$ hold.*

Proof. Since the public transportation network has to be entered and exited by a `LINK_EDGE` labeled edge, both a and $\overleftarrow{a}$ are access-node candidates according to our construction rules of the multi-modal graph (cf. Section3.5), thus, $a, \overleftarrow{a} \in A$.

Now consider the subpath $P' \subset P$ from s to $\overleftarrow{a}$. Then the path P' is also a shortest path. Furthermore, all nodes in P' after a belong to the public transportation network since P is conforming to L. Hence, it follows that $a \in A(s)$ by identifying b of Definition 10 with $\overleftarrow{a}$.

Regarding $\overleftarrow{a} \in \overleftarrow{A}(t)$, we consider the shortest t-s-path in $\overleftarrow{\boldsymbol{G}}$ which corresponds to the shortest s-t-path in G. The same argument is then applied to obtain $\overleftarrow{a} \in \overleftarrow{A}(t)$. □

We can now deduce the main correctness theorem for Access-Node Routing.

Theorem 5 (Correctness of Access-Node Routing)**.** *Access-Node Routing is correct.*

Given a Σ-labeled multi-modal graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$, source and target nodes $s, t \in \boldsymbol{V}$, a departure time τ, a regular language L having the enclose property, as well as forward and backward access-nodes $A(v)$ resp. $\overleftarrow{A}(v)$ conforming to L, applying the Access-Node Routing algorithm yields correct shortest paths from s to t wrt. the language L.

Proof. Let P be the shortest path in $\boldsymbol{G}$. If P is solely using the road network, then s and t are in the same connected component of the road network of $\boldsymbol{G}$. Hence, the locality filter $L(s, t)$ evaluates to true and the shortest path is found by the correctness of the applied uni-modal time-independent query algorithm for local queries.

So assume P is using the public transportation network. By Lemma 5 the shortest path uses access-nodes $a \in A(s)$ and $\overleftarrow{a} \in \overleftarrow{A}$ to enter resp. exit the public transportation network. Furthermore, between a and $\overleftarrow{a}$ only edges of the public transportation network are contained in P. Moreover, both a and $\overleftarrow{a}$ are contained in the sets of forward resp. backward access-nodes available to our query algorithm, which follows from Lemmas 3 and 4. Since our query algorithm computes shortest paths from all $A(s)$ to all $\overleftarrow{A}(t)$ and selects the minimum path in the end, the path from s to t over $a \in A(s)$ and $\overleftarrow{a} \in \overleftarrow{A}(t)$ is chosen.

We conclude that Access-Node Routing is correct. □

Theorem 6 (Correctness of Core-Based Access-Node Routing)**.** *Core-Based Access-Node Routing is correct.*

Given a Σ-labeled multi-modal graph $\boldsymbol{G}_C$ obtained by contracting $\boldsymbol{G}$, source and target nodes $s, t \in \boldsymbol{V}$, a departure time τ, a regular language L having the enclose property, as well as forward and backward access-nodes $A(v)$ resp. $\overleftarrow{A}(v)$ on $\boldsymbol{G}_{core} \subset \boldsymbol{G}_C$ conforming to L, applying the Access-Node Routing algorithm yields correct shortest paths from s to t wrt. the language L.

Proof. By Theorem 3 Core-Based Routing is correct. Furthermore, by Theorem 5 Access-Node Routing restricted to $\boldsymbol{G}_{\text{core}}$ is correct. Regarding the ambiguity of the paths toward/from the forward resp. backward access-nodes, the forward access-nodes are initialized with their keys set to the minimum distance from the source (see Equation 5.30). The distance labels at the core-exit-nodes are set according to the combined path of minimum length (see Equation 5.31). Thus, the shortest s-t-path is found and we conclude that Core-Based Access-Node Routing is correct. □

5.4.7. Discussion

In this section we presented a new speed-up technique called Access-Node Routing. Although, it only works with automata conforming to the enclose property (cf. Definition 12), i.e., the road network can be used in the beginning and the end alone, we believe this is sufficient, as using the car in the middle of the journey is undesirable in most cases anyway.

Access-Node Routing adopts some of the ideas of Transit Node Routing. We use access-nodes to jump in and out of the public transportation network by performing table-lookups. The actual query is then restricted to the public transportation network which only forms a small portion of our graphs.

We presented two approaches for computing access-nodes, an exact approach and an approach where we approximate the set of access-nodes. While the exact approach yields smaller sets $A(v)$ of access-nodes, it turns out to be too expensive to compute as the running time is too high since it requires full profile searches on $\boldsymbol{G}$. Thus, for our experiments we decided to use the approximate approach which only runs a profile search on the public transportation network. This makes it especially feasible for multi-modal graphs having a small public transportation network (e.g., the road network of Europe and America together with an intercontinental flight network).

Furthermore, we presented that Access-Node Routing can be combined with Core-Based Routing which requires access-nodes only for the much smaller core graph $\boldsymbol{G}_{\text{core}}$.

Regarding query times we gain a tremendous speed-up, as the largest parts of our

multi-modal graphs, namely the road networks, no longer have to be searched. Indeed, our experimental evaluation in Section 6.5 shows that query times are almost constant with respect to the size of the road networks. This is due to the fact that the search space almost only consists of public transportation network nodes. By these means, we are able to achieve query times of a few milliseconds on continental-sized graphs.

5.5. Summary

In this chapter we presented speed-up techniques for Dijkstra's algorithm. First, we presented basic ingredients which are an integral part of many of today's high-performance uni-modal speed-up techniques. We introduced every ingredient regarding time-independent uni-modal routing and augmented each to time-dependency and further to multi-modality.

Moreover, we presented three tailored multi-modal speed-up techniques: Core-Based Routing, Core-ALT and Access-Node Routing. These are the main results from this chapter.

- The basic ingredients for speeding up Dijkstra's algorithm are bi-directional search, goal-directed search (from which we presented ALT and Arc-Flags) and contraction.

- The augmentation to time-dependency is possible for every technique, however, it turns out very hard for bi-directional search and contraction. This is due to the fact that the arrival time at the target is not known in advance for which makes bi-directional search difficult. Since contraction uses bi-directional search during query, the same problems apply here.

- Regarding the augmentation to multi-modality, only Arc-Flags turns out difficult. This is due to the fact that paths are pruned during the query. Thus, paths which are unimportant during preprocessing are pruned during the query, although, they might become important when we use different automata. Computing non-trivial arc-flags that work with whole classes of automata remains an open challenge.

- Based on contraction, we developed a multi-modal speed-up technique called Core-Based Routing dividing the graph into a core and a component. We restrict the preprocessing to nodes in the road network whose incident edges all support the same modes of transportation (foot or car). This has two advantages. First, we only bypass time-independent parts of the network. Second, we ensure that

distances with respect to languages having the variable length property are preserved. This allows us to develop an efficient and general query algorithm. The problem of time-dependency during contraction does not occur here, since the time-dependent parts of the graph are fully contained in the core and, thus, the bi-directional search on the component is purely time-independent.

- Core-Based Routing by itself only yields mild speed-ups. Thus, we combined Core-Based Routing with multi-modal uni-directional ALT on the core called Core-ALT. While preprocessing is straightforward, as we simply restrict landmark and distance computation to the lower bound core graph $\underline{G_{\text{core}}}$, the integration of ALT into Core-Based Routing requires some extra consideration. Since potentials for ALT are computed with respect to the target node which, however, might not be part of the core, we have to introduce proxy-nodes to obtain a feasible potential function.

- While both Core-Based Routing and Core-ALT are 'general' in the sense that their only restriction is that they require automata having the variable length property, we relaxed the claim for generality further and introduced a new speed-up technique called Access-Node Routing. Its premise is that the road network is only used at the beginning and the end of the journey. Thus, for each node in the time-independent road network, we compute the relevant access-nodes into the public transportation network during preprocessing. The query is then restricted to the (time-dependent) public transportation network between the access-nodes of the source and target nodes, sparing the expensive search on the road network (which makes up the biggest part of our multi modal graphs). For local paths that do not use the public transportation network, we can use any uni-modal time-independent speed-up technique that can solve the task in a matter of microseconds.

- Access-Node Routing can be combined with the Core-Based Routing approach with only a few minor modifications to the query algorithm. This reduces the amount of precomputed data significantly, as access-nodes are only required for road network nodes of the core.

- Our main contribution from Access-Node Routing is that we are able to 'separate' the road network from the public transportation network. Thus, we can use different routing algorithms for local queries using the road network (e.g., a high-performance uni-modal speed-up technique) and the public transportation network. Not only is it possible to apply any multi-modal speed-up technique to the public transportation network but in fact, a different shortest path algorithm.

For example, we could use multi-criteria optimization on the public transportation networks to account for ticket fares, transfers, etc.

Chapter 6

Experiments

In this chapter we give an experimental evaluation. Besides examining the impact of the finite automaton used during query, we especially focus on the speed-up techniques Core-ALT and Access-Node Routing introduced in Chapter 5.

This chapter is organized as follows. First of all, we present the input data used throughout the experiments. In Section 6.3 we then compare the performance of the basic multi-modal routing algorithm using different finite automata.

In Section 6.4 we evaluate Core-ALT. We study different choices of the contraction parameters c and h affecting the amount of contraction. It turns out that speed-ups of Core-ALT highly depend on the finite automaton.

Finally, in Section 6.5 we run some tests on our new speed-up technique Access-Node Routing. It turns out that the query performance is almost constant with respect to the input size, as the query is restricted to the very small public transportation networks. We are able to perform intercontinental queries on a flight and road network of Europe and North America within 2.3 milliseconds.

The section is wrapped up by a summary of the main results.

6.1. Input

In this section we introduce the input data we use throughout our experiments. All networks are based on real world data and are, thus, not synthetic. However, not all aspects that are modeled into the graphs (i.e., check-in and check-out times at airports regarding our flight models), were available to us by the raw data. At these points we use synthetic data which we point out explicitly at the appropriate places.

We are not going into more detail about the raw data an its processing at this point. Please refer to Appendix B for details.

6.1.1. Graphs

Road Networks. As described in Appendix B.1, our raw data contains most of the Western European countries, the whole United States of America and Canada. The data is from 2006. Furthermore, the raw data allows us to generate subgraphs at a resolution of states resp. countries easily. The road networks make up the biggest part of our multi-modal graphs. Thus, we choose three different networks with increasing size: The German road network having approx. 4.5 million nodes and 11 million edges, the road network of whole Europe with approx. 30 million nodes and 72 million edges and finally a network consisting of North America and Western Europe having approx. 50 million nodes and 124.5 million edges. Note that this network is not connected, as it contains two separate continents. It will only be connected eventually by the overlaying flight network.

Railway Networks. Regarding the railway raw data, we had the timetables of the winter period from 1996/1997 and the winter period 2001/2002 at our disposal. While the former contains most long distance trains for Central Europe, the latter contains all trains for Germany that are operated by the Deutsche Bahn. However, we omit local commuter trains like the German 'S-Bahn'. We generate graphs according to the realistic time-dependent model with constant transfer times, i.e., the transfer time is constant for all trains passing through the same station. Transfer times are also available through the raw data. We basically generate two graphs. One for Germany using the timetable from the winter period 2001/2002 (without the commuter trains) and one for whole Europe using the timetable of the winter period 1996/1997 containing only long distance trains.

Flight Networks. Our flight networks are not based on real 'raw' timetable data but, instead, are taken from publicly available timetables from the Internet (cf. Appendix B.3). We created converters for the public timetables of two major flight alliances containing many airlines: Star Alliance [Sta97] and Oneworld [One99]. The data of Star Alliance is from the period September 1st to November 16th 2008 containing 965 airports and 21 084 flights, while the data of Oneworld is from the period between October 3rd and October 31st containing 621 airports and 8 796 flights. These timetables are composed into the flight class model whereas each flight alliance implies one distinct flight class. Unfortunately, real values for check-in, check-out and transfer times were not available to us. Thus, we use the very conservative values of 120 minutes, 90 minutes and 60 minutes, respectively. Transfers between different airlines are set to 150 minutes.

Table 6.1.: Size of our main input graphs broken down into their components.

Network	No. of Nodes	No. of Edges	No. of Points
`de-road-rail-flight`	4 804 664	11 603 003	725 113
road network	4 692 524	11 279 784	—
railway network	108 453	296 330	693 012
flight network	3 687	13 261	32 101
`europe-road-rail-flight`	30 722 060	74 279 064	2 622 109
road network	30 203 343	72 799 049	—
railway network	515 024	1 411 867	2 589 989
flight network	3 693	13 274	32 120
`na-eur-road-rail-flight`	50 700 647	125 939 503	2 622 610
road network	50 181 903	124 458 952	—
railway network	515 024	1 411 867	2 589 989
flight network	3 720	13 398	32 621

Multi-Modal Graphs. Our multi-modal graphs are combinations of the uni-modal graphs introduced above. We generate three main graphs with increasing size.

- `de-road-rail-flight`.
  This graph consists of the German road network plus the German railway network from the timetable of the winter period 2001/2002 as well as the flight network of both flight alliances.

- `europe-road-rail-flight`.
  This graph uses the road network of Europe plus the long distance railway network of Europe from the winter period 1996/1997 and the flight network of both flight alliances.

- `na-eur-road-rail-flight`.
  This is our largest instance. It consists of the European and North American road network plus the European railway network and the flight network of both flight alliances.

From each of these networks we compute the strong connected component. Figures for the resulting graph sizes are shown in Table 6.1 broken down into the sizes of the subnetworks (road, rail and flight). Note that the number of edges from the subnetworks may not exactly sum up to the number of edges of the resulting multi-modal graph, as there are additional `LINK_EDGE` labeled edges inserted when combining the networks.

6.1.2. Automata

Multi-modal queries require a finite automaton as input. In Section 4.3.2 on page 56 we already introduced a number of automata that are interesting for shortest path queries. These are also used as input for our experiment we, thus, recall them only briefly.

- `car`.
  This is the simplest of all automata. It consists of only one state (that is both final and initial) and one loop composed of a `CAR_EDGE` labeled transition. It can be seen as the equivalent automaton for conduction road queries on uni-modal networks. This is useful for comparing the performance of the multi-modal routing algorithm against the performance of the classic uni-modal Dijkstra algorithm.

- `everything`.
  This automaton allows arbitrary shortest paths by representing the language $L = \Sigma^*$. The transition graph is shown in Figure 4.3a on page 4.3a.

- `foot-and-rail`.
  This automaton is a hierarchical automaton which restricts shortest paths to the road and railway networks. The language accepted by this automaton is $L = f^*lr^*lf^*$, where $f =$ `FOOT_EDGE`, $l =$ `LINK_EDGE` and $r =$ `RAIL_EDGE`. The transition graph is shown in Figure 4.3b. We use this automaton only on the German network.

- `car-and-flight`.
  This is the equivalent of the `foot-and-rail` automaton used on the Europe and the continental networks. Its accepted language is $L = c^*lg^*lc^*$, where $c =$ `CAR_EDGE` and $g =$ `FLIGHT_EDGE`. The transition graph is shown in Figure 4.3c.

- `everything-reasonable`.
  This is our largest automaton and implements a reasonable usage of all means of transportation in a hierarchical approach. The transition graph is shown in Figure 4.4 on page 57.

In Section 6.3 we compare the performance of the multi-modal query algorithm when used with different automata. Subsequent experiments are then restricted to the automata `everything-reasonable` and `foot-and-rail` resp. `car-andflight`.

6.2. Experimental Setup

Our implementation of all our algorithms is done in C++ solely based on the STL and the Boost library at a few rare places. For efficiency, we do not use virtual methods and class inheritance but rather make excessive use of templates. The way we store graphs is done using a binary format consisting of a *forward star* (adjacency array) representation which is very efficiently implemented. For that reason, we can handle huge graphs very well with a minimum of required memory—the performance of reading a graph from file is only limited by the hard drive speed. The setup of our main data structures is presented in Appendix A in more detail.

As C++ compiler we use GCC version 4.3.1 on SuSe Linux 11.0 (kernel version 2.6.22.17) with the flags `-O3 -ffast-math -fomit-frame-pointer -funroll-loops -DNDEBUG`.

Our experiments were conducted on a Dual-Core AMD Opteron 2218 processor having 2.6 GHz, 1 MiB level 2 cache (each core) and 32 GiB of main memory. All of our programs are single threaded and, thus, only one of the cores is used.

In our experiments on speed-up techniques we report two aspects. Regarding the preprocessing, we show the number of bytes per node additionally required due to the preprocessed data. On Core-Based speed-up techniques we charge the space consumption of the preprocessed data, although only computed on the small core graph, against the whole input graph. Regarding queries, we report the number of settled nodes and relaxed edges which is an indication of the size of the search space. In the case of multi-modal routing algorithms, we mean implicitly computed product-nodes (and edges). Furthermore, we report the average query time of single queries. By comparing the size of the search space with the query time we are able to deduce the algorithmic overhead.

The values regarding queries are computed by running a number of *random queries*, i.e., we pick two nodes s and t and a departure time $\tau < \Pi$ at random (For the 'small' graphs like Germany we use 1000 random queries, on the 'larger' graphs we only run 100 random queries). The number of settled nodes, relaxed edges and the query times are then reported as the average value over all random queries. When comparing different query algorithms on the same pairs of graph and automata, we use the same set of queries for each algorithm. Speed-ups are in terms of running time and are always relative to the basic multi-modal query algorithm as of Algorithm 5 on page 5.

Table 6.2.: Comparison of query performance of the basic multi-modal query algorithm against different finite automata.

	de-road-rail-flight		
Automaton	Relaxed Edges	Settled Nodes	Time [s]
car	10 896 903	2 323 641	2.43
everything	9 836 137	2 114 948	2.60
foot-and-rail	10 622 387	2 247 428	3.69
everything-reas.	29 342 126	6 240 651	10.17

	europe-road-rail-flight			na-eur-road-rail-flight		
Automaton	Relaxed Edges	Settled Nodes	Time [s]	Relaxed Edges	Settled Nodes	time [s]
car	70 488 720	15 014 721	18.06	67 741 290	14 156 302	16.74
everything	69 949 157	15 045 051	23.39	108 677 456	22 672 284	39.50
car-and-flight	123 063 212	26 184 714	36.84	169 075 629	35 155 882	45.43
everything-reas.	150 594 892	32 089 210	60.57	213 882 663	44 599 766	87.32

6.3. Multi-Modal Routing

In this section we examine the performance of our basic multi-modal query Algorithm (see Algorithm 5 on page 5). As worked out, the algorithm computes the product network $G^{\times}$ between the graph G and the transition graph of the finite automaton implicitly. Hence, we expect that the performance of the algorithm does not depend on the input graph but also on the automaton used.

Table 6.3 reports query performance on the three multi-modal graphs `de-road-rail-flight`, `europe-road-rail-flight` and `na-eur-road-rail-flight`. We observe that query performance degrades with increasing complexity of the finite automaton. The `everything`-automaton has only one state, hence, the number of settled nodes and relaxed edges is about 50% of the original graph size. However, in all graphs the 2nd automaton (`foot-and-rail` and `car-and-flight`, each having three states), does not deteriorate performance by a factor of three. This is explained by the fact that not every arbitrary path is allowed (only such paths that conform to the automaton) which limits the number of product-nodes and relaxed edges, thus, only yielding a mild decrease in query performance compared to the `everything`-automaton.

Table 6.3 compares the performance of a classic uni-modal uni-directional and time-independent Dijkstra algorithm against our multi-modal implementation on two graphs: The German road network and the road network of Europe. We omit the continental network since when using roads alone, most of the queries would fail since

Table 6.3.: Comparison between uni-modal and multi-modal routing on two road networks.

	de-road			europe-road		
Algorithm	Relaxed Edges	Settled Nodes	Time [s]	Relaxed Edges	Settled Nodes	time [s]
Uni-modal Dijkstra	5 527 565	2 398 863	1.31	35 551 637	15 494 159	10.17
Multi-modal algorithm	11 385 495	2 398 863	2.05	73 148 401	15 494 159	15.97

there is no connection between the two continents. Because the road network contains both foot and car weights, we use the simple `car`-automaton on the multi-modal network. The uni-modal Dijkstra algorithm is restricted to car weights. As we can see in Table 6.3, our multi-modal algorithm slows down the query time by approx. 50%. This is due to the more complex data structures (we have to use product-nodes instead of simple nodes throughout the algorithm) and also due to the fact that for each outgoing edge a transition in the automaton has to be performed.

Regarding the number of relaxed edges, our multi-modal query algorithm relaxes two edges per road-edge: One `CAR_EDGE` labeled edge and the same edge with the label `FOOT_EDGE`. However, the foot edges are not processed further, since the `car`-automaton does not allow `FOOT_EDGE` labeled transitions.

6.4. Core-ALT

In this section we focus on Core-Based Routing, especially with respect to Core-ALT (see Section 5.3). Recall that we only contract the road-network, and within the road network only nodes where all incident edges have the same modes of transportation available (car or foot). Thus, the time-dependent parts of the network are contained inside the core.

Preprocessing for Core-Based Routing (see Section 5.2.1) allows for two parameters. The contraction rate c and the maximal number of nodes that are allowed to be bypassed by a shortcut, h (hop-count). More aggressive contraction through c demands for longer shortcuts. Thus, both values of c and h have to be adjusted together in order to yield reasonable results. We only focus on multi-modal Core-ALT here. Experiments on uni-modal graphs with Core-ALT (both time-independent and time-dependent) are conducted in [Del09a] yielding speed-ups up to 3 000 and 700 in the time-independent and time-dependent case, respectively.

Landmarks are computed on the lower bound graphs of the respective cores using 64 avoid landmarks.

We test the performance of Core-ALT on the graph of `europe-road-rail-flight`

Table 6.4.: Core-ALT on `europe-road-rail-flight` with varying c and h values during contraction and three different automata: `car`, `car-and-flight` (`c.-&-f.`) and `everything-reasonable` (`evr.-reas.`). The reference values are computed by the basic multi-modal query algorithm without any speed-up technique on the original graph (cf. Table6.3). Landmark distances for ALT are computed using 64 avoid landmarks.

			Preprocessing				Query		
Automaton	c	h	Core Nodes	$\|E\|$-incr.	Time [min]	Space [B/n]	Settled Nodes	Time [s]	Speed-up
`car`	—	—	—	—	—	—	15 014 721	18.0	1.0
	0.5	10	35.9%	11.5%	50	180.9	3 864 000	4.90	3.68
	1.0	20	18.2%	16.9%	45	93.2	1 880 653	2.66	6.78
	2.0	30	15.7%	18.7%	44	80.8	1 589 217	2.24	8.06
	2.5	40	15.1%	19.3%	46	77.7	1 596 875	2.31	7.81
	3.0	50	14.8%	20.1%	49	76.1	1 505 732	2.35	7.68
	3.5	60	14.8%	20.7%	50	75.7	1 502 848	2.21	8.17
	4.0	70	14.6%	21.4%	52	75.0	1 530 528	2.30	7.85
	5.0	100	14.4%	22.8%	59	74.0	1 475 097	2.29	7.88
`c.-&-f.`	—	—	—	—	—	—	26 184 714	36.8	1.0
	0.5	10	35.9%	11.5%	50	180.9	13 510 833	17.2	2.14
	1.0	20	18.2%	16.9%	45	93.2	5 438 549	7.56	4.87
	2.0	30	15.7%	18.7%	44	80.8	4 513 729	6.57	5.6
	2.5	40	15.1%	19.3%	46	77.7	4 238 075	6.41	5.74
	3.0	50	14.8%	20.1%	49	76.1	4 222 589	6.44	5.72
	3.5	60	14.8%	20.7%	50	75.7	4 150 769	6.36	5.79
	4.0	70	14.6%	21.4%	52	75.0	4 005 002	6.27	5.87
	5.0	100	14.4%	22.8%	59	74.0	4 052 246	6.52	5.65
`evr.-reas.`	—	—	—	—	—	—	32 089 210	60.5	1.0
	0.5	10	35.9%	11.5%	50	180.9	38 660 857	75.8	0.79
	1.0	20	18.2%	16.9%	45	93.2	18 748 596	39.0	1.55
	2.0	30	15.7%	18.7%	44	80.8	16 053 889	34.9	1.73
	2.5	40	15.1%	19.3%	46	77.7	15 455 441	34.2	1.76
	3.0	50	14.8%	20.1%	49	76.1	15 129 244	34.2	1.76
	3.5	60	14.8%	20.7%	50	75.7	15 044 912	34.4	1.76
	4.0	70	14.6%	21.4%	52	75.0	14 921 469	34.7	1.74
	5.0	100	14.4%	22.8%	59	74.0	14 719 448	35.6	1.68

using the automata `car`, `car-and-flight` and `everything-reasonable`. We omit the `everything` automaton as it is rather unrealistic since it allows arbitrary paths in the network. Table 6.4 reports our results for varying c and h values.

Preprocessing. Regarding preprocessing, we report the size of the core in relation to the size of the original graph regarding the number of nodes. Furthermore, we report the increase in edges compared to the original graph which accounts for the shortcuts that are inserted into the core.

While in pure uni-modal road networks, the core can be shrunk below 1% of the graph size (see [Delo9a]), this is not possible with our contraction routine. This is due to our restrictions regarding bypassable nodes. Hence, from $c = 3.5$ and $h = 50$ upward, the contraction routine is satured in the sense that increasing c and h further shows no improvement.

The additional space required by the landmarks and their distances levels off at approximately $75\,\mathrm{B/n}$ and correlates directly with the size of the core since landmark distances are only computed on the core.

Query. Regarding query performance, we observe that speed-ups heavily depend on the complexity of the automaton. While using the basic `car`-automaton yields speed-ups of 8.17, the best speed-up achievable through the slightly more complex `car-and-flight`-automaton (`c.-&-f.`) drops down to 5.87. This is due to two reasons. First, allowing less modes of transportation sort of prunes more parts of the core. Note that from $c = 3.0$ and $h = 50$ upward, almost all bypassable nodes of the road network are extracted from the core. This virtually has the effect of 'shrinking' the core. Second, landmark distances are computed on the lower-bound graph $\underline{G_{\text{core}}}$ of the core. While the lower-bounds are exact for road network edges, they are very bad on public transportation edges. For example, if there is only one flight per day on a flight-edge taking one hour, the lower bound of that edge is one hour, though, the real weight can be up to 23 hours. This yields a very bad potential function.

Furthermore, it should be noted that relaxing very long shortcuts (as they appear in the core) into the 'wrong direction' is counterproductive, as the algorithm starts settling many nodes in parts of the graph it would not have reached without the shortcuts. This is reflected by the speed-up of 0.79 with the `everything-reasonable` automaton when setting low values of $c = 0.5$ and $h = 10$.

We conclude that contraction parameters of $c = 3.0$ and $h = 50$ are probably sufficient for our multi-modal Core-Based routing approach as the improvements regarding space consumption and query performance for higher value are almost negligible. Moreover, Core-ALT performance depends on the automaton used. As soon as

the public transportation network is taken into account, potentials become worse as speed-ups decrease.

6.5. Access-Node Routing

This section covers experiments regarding our newly developed speed-up technique Access-Node Routing (cf. Section 5.4). Access-Node Routing is based on the idea that for each node of the road network there is a limited number of relevant access-nodes that are used to enter (forward access-nodes) resp. exit (backward access-nodes) the public transportation network. Assuming we only want to use the road network in the beginning and the end of the journey, we precompute for each node in the road network their forward and backward access-nodes together with their distances and reduce the query to the public transportation network.

We presented two approaches to compute access-nodes: An exact variant and a variant that only computes approximate access-nodes. In the subsequent experiments we always use the approximate version since the exact variant is far too slow to be executed in reasonable time. The approximate approach computes for each access-node candidate a full backward profile search on the public transportation network and then grows backward shortest path trees simultaneously from all access-node candidates that were reached by the backward profile search using their upper bounds as initial keys.

Input Data. While approximate access-nodes are faster to compute, they still take too long for our big public transportation networks in our graphs. Thus, we conduct our experiments on the following multi-modal graphs with reduced public transportation networks.

- `de-road-rail(long)`.
  This is the road network of Germany together with all long distance trains from the timetable of the winter period 2001/2002. This includes InterRegio (IR), InterCity (IC) and InterCityExpress (ICE) trains. We use this graph together with the `foot-and-rail` automaton.

- `ny-de-road-flight`.
  This is a small intercontinental network composed of the U.S. state of New York and Germany. The German part of the graph has 4 692 524 nodes, whereas the New York part only consists of 611 024 nodes. Both components are connected through the flight network of both our flight alliances. We use this graph together with the `car-and-flight` automaton.

Table 6.5.: Preprocessing Figures for (Core-Based) Access-Node Routing. We report the number of access-node candidates, the average number of forward and backward access-nodes per road node and the preprocessing time for computing access-nodes as well as the additionally required space per node.

Network	Core-Based	AN-Cand.	Forward Access-Nodes	Backward Access-Nodes	Time [min]	Space [B/n]
de-road-rail(long)		473	32.4 (6.8%)	20.9 (4.4%)	143	435.2
de-road-rail(long)	✓	473	31.0 (6.5%)	19.7 (4.1%)	26	55.6
ny-de-road-flight	✓	26	14.2 (54.6%)	14.2 (54.6%)	< 1	30.5
na-eur-road-flight	✓	359	118.7 (33.0%)	119.1 (33.1%)	161	223.5

- na-eur-road-flight.
  This is our largest instance for Access-Node Routing. It consists of the road network of both continents: Europe and North America (including Canada) (see Table 6.1 for figures) and the flight network of both flight alliances. Again, we use this graph together with the car-and-flight automaton.

For Core-Based Access-Node Routing these graphs are contracted with parameters $c = 2.5$ and $h = 40$ on de-road-rail(long) and $c = 4.0$ and $h = 70$ on both ny-de-road-flight and na-eur-road-flight.

Preprocessing. The first experiment is devoted to (approximate) preprocessing of Access-Nodes. Table 6.5 reports figures for the graph instances from above. On all graphs (except de-road-rail(long)), we use the Core-Based Access-Node Routing approach where we first contract the input graph and reduce the access-nodes computation to the much smaller core graph (cf. Section 5.4.5). Regarding de-road-rail(long), the average number of forward and backward access-nodes per road node is 32.4 resp. 20.9. Thus, 32.4 railway stations are important (on average) to enter the railway network. While this seems a very high number (even more if we consider that these railway stations have to be reached by foot), there are two reasons for this. First, the railway network is sparsely embedded into the road network, thus, for a single road network node a lot of stations are important at least once a day. Second, long distance trains do not operate very frequently on some parts of the network. As a consequence, the upper bounds carry much weight during preprocessing, yielding bad approximations which leads to many unnecessary access-nodes.

The same effect can be observed in both the ny-de-road-flight and the na-eur-road-flight network, as flights are even more infrequent. This makes it attractive to cover far distances by car[1], thus including many (also far away) airports into the set of

[1]Imagine living near Frankfurt Airport, but a flight to New York from Frankfurt Airport requires us to

Table 6.6.: Query performance of (Core-Based) Access-Node Routing without local queries (i.e., all shortest paths use the transportation network) compared to plain multi-modal DIJKSTRA.

	DIJKSTRA		Access-Node Routing			
Network	Settled Nodes	Time [ms]	Core-Based	Settled Nodes	Time [ms]	Speed-up
de-road-rail(long)	2 483 030	3 491.7		13 779	4.7	742.9
de-road-rail(long)	2 483 030	3 491.7	✓	14 017	6.1	572.4
ny-de-road-flight	9 155 893	9 253.5	✓	4 074	1.9	4 870.2
na-eur-road-flight	46 244 703	72 566.3	✓	4 337	2.3	31 550.5

relevant access-nodes.

Preprocessing times are in the range of several hours (between 26 minutes for the core of the German network and almost three hours for the core of the continental network). Note that regarding de-road-rail(long), switching to Core-Based Access-Node Routing drastically reduces both preprocessing time and the required space for the access-nodes, as we restrict ourselves to the core graph. Furthermore, the preprocessing time is bounded by the size of the public transportation network. On the core of de-road-rail(long) it takes 26 minutes to compute access-nodes, while on the core of ny-de-road-flight with the extremely small flight network it takes below 1 minute (although the road network is larger here).

Query. Regarding the query performance, we expect a massive speed-up, as the biggest part of the shortest path search, namely the road network, is absent. Table 6.6 reports figures for queries using (Core-Based) Access-Node Routing. We run 1 000 random queries on the small de-road-rail(long) and ny-de-road-flight networks whereas we run 100 random queries on the large na-eur-road-flight network. Note that all shortest paths in this experiment utilize the public transportation network. Thus, we solely analyze the performance of Access-Node Routing. Since the graphs are slightly different from the ones used in Table 6.3, we also report the performance of the simple multi-modal routing algorithm (on the original graph) to which speed-ups are compared. The number of settled nodes only refers to the search in the public transportation network plus the bi-directional search on the component in phase one of the core-based query algorithm.

We observe a drastic drop in both the number of settled nodes and the query time when using Access-Node Routing. In de-road-rail(long) we observe that the query

wait for several hours at the airport. During this time we can reach a lot of other airports in Germany by car from which there might be a flight to New York that reaches the destination earlier.

Table 6.7.: In-depth analysis of *Core-Based* Access-Node Routing. This table reports the distribution of query time among the particular phases of the query algorithm: The bi-directional search on the component using time-independent road network edges, the table-lookups regarding the access-nodes of all core-entry- and core-exit-nodes and the search on the public transportation network. Furthermore, we report the amount of local queries (paths that do not use the transportation network) when generating 1000 (`de-road-rail(long)`, `ny-de-road-flight`) and 100 (`na-eur-road-flight`) random queries.

Network	Comp.-Query	AN-Lookup	Public-Transport.	Total [ms]	Local Queries
`de-road-rail(long)`	0.15 (2.4%)	0.08 (1.4%)	5.87 (96.2%)	6.1	2.3%
`ny-de-road-flight`	0.35 (18.4%)	0.01 (1.1%)	1.53 (80.5%)	1.9	80.5%
`na-eur-road-flight`	0.42 (18.2%)	0.18 (7.9%)	1.70 (73.9%)	2.3	24%

time increases slightly from 4.7 to 6.1 milliseconds when switching to the core-based variant. This is due to the additional overhead of the Core-Based Routing algorithm (The bi-directional search on the component has to be conducted and the computation of the forward and backward access-nodes is slightly more complicated since we have multiple core-entry- and core-exit-product-nodes). The performance of Access-Node Routing on `ny-de-road-flight` beats the performance on `de-road-rail(long)`. This is due to the fact that the flight network is significantly smaller than the railway network in `de-road-rail(long)`. Moreover, the number of access-nodes per road node is only 14.2 whereas in Germany it is twice that much.

The highest speed-up of 31 550.5 is achieved on the continental-sized road network of Europe and North America. We are able to perform intercontinental queries with an average time of 2.3 milliseconds compared to over 72 seconds when the standard algorithm. The very similar query time compared to `ny-de-road-flight` is due to the fact that the flight network in both graphs is of the same size. The additional time of 0.4 milliseconds, however, is mostly spent on looking up the many more access-nodes on `na-eur-road-flight`.

The query algorithm for Core-Based Access-Node Routing is made up of three distinct phases (see Figure 5.15 on page 109). The first phase involves the bi-directional time-independent search on the component until all core-entry- and core-exit-nodes are discovered. Next, for each core-entry- resp. core-exit-node we look up all access-nodes with their correct distances. The third phase finally involves the multi-modal search on the public transportation network. Table 6.7 reports the distribution of the running time among the particular phases of the query algorithm. It is clearly seen that the public transportation query makes up the major part of the running time

(between 73.9% and 96.2% depending on the network). The time for looking up the access-nodes is insignificant as it is less than 2% on `de-road-rail(long)` and `ny-de-road-flight`. Only on `na-eur-road-flight` looking up the access-nodes takes 7.9% of the running time. This is due to the fact that the number of average access-nodes per road node is significantly higher than on the other two networks (cf. Table 6.5).

Furthermore, we report the relative number of local queries. These figures are obtained by computing 1000 random queries (100 on `na-eur-road-flight`) and counting those queries that do not use the public transportation network. We observe a great variation depending on the network. While in `de-road-rail(long)` only 2.3% of the queries do not use the railway, in `ny-de-road-flight` 80.5% of the queries are local. This is due to the fact that in `de-road-rail(long)` it almost never pays off to go the whole route by foot. On the other side, in `ny-de-road-flight` the two components (Germany and the state of New York) are very uneven (the road graph of New York only contains only approx. 600 000 nodes). Thus, a lot of queries are not intercontinental. Moreover within Germany and New York it rarely pays off to use an airplane as distances covered are not far enough. While for `na-eur-road-flight` the two components of the road network are more balanced, average distances within Europe resp. North America are far enough that it often pays off to take a flight. Hence, the amount of local queries drops to 24%.

6.6. Summary

In this section we conducted experiments on multi-modal graphs and presented their results. All experiments were based on multi-modal networks obtained from real world data (see also Appendix B). These are the main results.

Multi-Modal Routing.

- Query performance drops about 50% when substituting uni-modal Dijkstra with the multi-modal routing algorithm (See experiment in Table 6.3 and Algorithm 5 on page 55).

- The performance of multi-modal queries not only depends on the graph size but also on the complexity of the finite automaton (See experiment in Table 6.3). For example, on the `na-eur-road-rail-flight` graph with the simplest `car` automaton we achieve a query performance of 16.74 sec on average, whereas the complex `everything-reasonable` automaton yields queries of 87.32 sec.

Core-ALT.

- Regarding Core-Based Routing and Core-ALT, the choice of the contraction parameters c and h influences the amount of contraction on the core graph. However, since we do not bypass the public transportation networks, we are not able to achieve as high contraction ratios as it can be done with pure uni-modal contraction on road networks. From $c = 3.5$ and $h = 50$ upward, we observe no further improvements as the core size levels off at approximately 15% on the `europe-road-rail-flight` graph (cf. Table 6.4).
- Query performance of Core-ALT, again, depends on the complexity of the finite automaton. With the simple `car` automaton we achieve speed-ups of 8.17 on `europe-road-rail-flight`, whereas using the most complex `everything-reasonable` automaton only yields very mild speed-ups of 1.76 (see Table 6.4). Moreover, speed-ups are generally much smaller than in uni-modal road networks, as the quality of the potential functions of the ALT algorithm degrades when public transportation networks are involved.

Access-Node Routing.

- The time for computing (approximate) access-nodes is dominated by the size of the public transportation network. On the `ny-de-road-flight` graph with a very small public transportation network, we need less than one minute for preprocessing. On the other hand, on the `de-road-rail(long)` graph of similar size but including a much bigger public transportation network it takes 26 minutes for computing the access-nodes (cf. Table 6.5).
- Access-Node Routing can be restricted to the core graph obtained by the Core-Based Routing approach which reduces both preprocessing time and the required space per node significantly. Thus, Core-Based Access-Node Routing is the preferred approach for large scale networks (also see Table 6.5).
- Query performance mainly depends on the size of the public transportation network. Both networks `ny-de-road-flight` and `na-eur-road-flight` yield similar query times of around 2 ms, although the latter graph is almost 10 times larger. Even the most complex public transportation network of the `de-road-rail(long)` graph yields query times under 10 ms (see Table 6.6).
- We conclude that with Core-Based Access-Node Routing we are able to compute intercontinental multi-modal queries on the biggest of our graphs in 2.3 ms time.

Chapter 7

Conclusion

In this work we dealt with multi-modal route planning. We set ourselves the goal to perform realistic multi-modal queries on large scale networks involving roads, railways and flights efficiently. The main results of this thesis are summarized in the following.

Models. We successfully combined the different realistic models for road, railway and flight networks into multi-modal graphs. Thereby, our road networks are time-independent and consist of car and foot edge weights. Our railway networks are based on the realistic time-dependent model with constant transfer times. Regarding the flight model, we worked out that the time-dependent railway model is not suited well to model flight timetables as the resulting graphs become unnecessarily large. Thus, we proposed a set of new efficient flight models from which we use the flight class model for our flight networks.

Basic Routing. In this work we were interested in solving the Earliest Arrival Problem, i.e., for some source and target nodes and a departure time find the route that arrives at the target first. In uni-modal scenarios the Earliest Arrival Problem is equivalent to the Shortest Path Problem on a weighted directed graph. Dijkstra's algorithm can be used to solve both time-independent, time-dependent and profile queries. We worked out that the simple Shortest Path Problem is not suited for multi-modal networks, as the shortest path may be undesirable (recall that it could force us to use the car between two trains in the middle of the journey). Thus, we augmented the Shortest Path Problem to involve constraints yielding the Label Constrained Shortest Path Problem. When restricted to regular languages L, the Label Constrained Shortest Path Problem can be solved by an augmentation of

DIJKSTRA's algorithm involving the product network between the multi-modal graph G and the transition graph of a finite automaton representing L. We presented an improved version of the algorithm where the product network is computed implicitly during the algorithm, resulting in significantly less space consumption.

In our experiments we observe a decrease of about 50% regarding query performance when using our multi-modal algorithm instead of plain uni-modal DIJKSTRA. Furthermore, the finite automaton used during query has a direct influence on query performance. On the world sized network average query times are between 16 and 87 seconds using the very simple `car` automaton and the most complex `everything-reasonable` automaton, respectively.

Adapting Ingredients for Speed-Up Techniques. We performed a survey of basic ingredients of today's speed-up techniques deriving from road networks. These are bi-directional routing, goal-directed search (in particular ALT and Arc-Flags) and contraction. We showed that all of these ingredients can be adapted to multi-modality easily with the exception of Arc-Flags. The problem concerning Arc-Flags seems to be the pruning of paths during preprocessing that may, however, become important for queries using a different automaton.

Core-Based Routing and Core-ALT. Based on contraction we presented a multi-modal variant of Core-Based Routing. We apply contraction only to the road network and therein only to nodes where all incident edges allow the exact same modes of transportation (foot and car). By these means the component is completely time-independent and the public transportation networks are fully contained in the core. This allows a relatively simple query algorithm as we can apply bi-directional search on the component without further considerations. Furthermore, a different multi-modal algorithm can be applied to the core graph independently. As an example we present Core-ALT where we use uni-directional multi-modal ALT on the core yielding a robust and general speed-up technique that works together with all 'reasonable' automata.

In our experiments we observe that our graphs cannot be contracted as aggressively as in uni-modal road-networks which is due to the restrictions as to which nodes are classified bypassable. Furthermore, speed-ups using Core-ALT are rather mild and lie between 8.1 and 1.7 when using the simple `car` and the complex `everything-reasonable` automata, respectively.

Access-Node Routing Our main contribution in this thesis is Access-Node Routing. Based on some ideas of Transit-Node Routing, we are able to compute entry- and exit-

points to the public transportation network directly (access-nodes). With the restriction that the road network is only used in the beginning and the end of journeys, we are able to isolate the search in the 'hard' public transportation network from the 'easy' road network, as we directly jump into the public transportation network from the source and target nodes of our query. Thus, the multi-modal shortest path search is restricted to the public transportation network. Regarding local queries that do not use the transportation network for shortest paths, we can use one of today's high-performance speed-up techniques for time-independent road networks which are able to solve this task in microseconds time. Since the public transportation networks are small compared to the road networks we already gain a dramatic improvement regarding query time, even when no further speed-up technique is used on the public transportation network.

Since the amount of preprocessed data for plain Access-Node Routing is quite high, combining Access-Node Routing with Core-Based Routing reduces the computation of the access-nodes to the much smaller core graph making this approach more feasible.

Experiments show that using Core-Based Access-Node Routing on a large scale network composed of Europe and North America together with a flight network, we are able to perform intercontinental queries in only 2.3 milliseconds time—even without applying any speed-up technique on the transportation network. Thereby, preprocessing is performed in under three hours time yielding space consumption of 223.5 bytes per node.

Summary. When switching from time-independent to time-dependent routing one can observe the effect that accelerating Dijkstra's algorithm becomes a much harder task. A similar effect is observed when switching to multi-modal routing. With an average query time of 87 ms on our largest graphs the demand for faster algorithms is even higher in multi-modal routing. With our main contribution Access-Node Routing we are, however, able to somewhat counteract the effect of increasing complexity as we separate the road network from the public transportation networks. Due to the modular design of our technique, we are able to apply query algorithms on the public transportation network and the road network independently. By these means local queries in the road network are very cheap. Furthermore, because of the small size of the public transportation networks we already gain speed-ups in the magnitude of 30 000 by solely applying a plain multi-modal Dijkstra algorithm on the public transportation network.

7.1. Future Work

The research field of multi-modal routing is still widely unexplored. Only few studies have been conducted regarding the Label Constrained Shortest Path Problem. In [BJM00] the problem has been theoretically examined yielding the result that when using regular languages on the labels, the problem is solvable in polynomial time. Regular languages seem to be sufficient for modeling shortest path constraints, thus, using finite automata together with our augmentation of Dijkstra's algorithm is feasible. The only experimental results obtained so far besides this thesis seem to come from [BJM00, Holo8, BBH+09].

However, we think multi-modal route planning is an important problem relevant to practical use. Hence, it deserves further attention. In the following we list some ideas for directing research on this topic.

Contraction Hierarchies. Contraction Hierarchies is a contraction based speed-up technique that applies the node-reduction and edge-reduction routines subsequently multiple times. We are truly interested in an adaption of Contraction Hierarchies to multi-modal routing. Especially, we are interested in a combination of Contraction Hierarchies with Access-Node Routing, as we believe that it is possible to drop the demand for locality filters, as 'local' queries using solely the road network might be computed implicitly with a small penalty concerning query time performance.

Incorporating Pedestrians. In our road network model we use a very basic approach for modeling pedestrians. We simply assume for every road segment a constant weight obtained from an average walking speed. We could, however, think of more realistic models regarding foot networks that also involve aspects like traffic lights and crosswalks. In the context of multi-modal routing this could be especially useful for urban transport companies. Having a combined pedestrian and public transportation network of a city, we could answer point-to-point queries resulting in a route that utilizes both the public transportation network and the pedestrian network yielding a 'convenient' path with a minimum number of street crossings or traffic light usage on the way to the bus or metro.

Accelerating Public Transportation Queries. In this thesis we worked out how we are able to 'separate' the road network from the public transportation network by using Access-Node Routing. This allows us to apply different speed-up techniques on local queries in the road network and long distance queries on the public transportation network. For our feasibility study we did not really care about the performance on

the public transportation and simply applied plain Dijkstra. Experiments on public transportation networks have shown that speed-up techniques are harder to adapt than one might expect. So we are always interested in further research on speed-up techniques for public transportation networks, both regarding to uni-modal and multi-modal public transportation networks.

Arc-Flags. In Section 5.1.3 we worked out that Arc-Flags is hard to adapt to multi-modal routing. Regarding this topic we would be interested in theoretical research, especially regarding the open Problems 1 and 2 on page 75. As Arc-Flags is an ingredient to the fastest known speed-up techniques of today, being able to use Arc-Flags for multi-modal routing may yield another approach for gaining nice speed-ups.

Profile Searches. As in our multi-modal networks the public transportation parts are time-dependent, we are interested in efficient ways of computing profile searches. In our scenarios using time queries, the user of the route planning system always has to state some departure time τ in advance. However, it would be much more convenient to be presented with all 'optimal' routes during the day from which the user can choose the route suiting him best. These 'optimal' routes depart exactly at those times of day where the profile function has its local minimums. Furthermore, our preprocessing algorithm for Access-Node Routing requires profile searches. Thus, we are interested in more efficient algorithms for computing profile queries.

Locality Filter for Access-Node Routing. Regarding Access-Node Routing we have to compute a local search on the road network, if the shortest path does not use the public transportation network. In our approach presented, we conduct the local search every time the source and target node are in the same connected component of the graph. We argued that the query on the road network is cheap as today's time-independent uni-modal speed-up techniques on road networks are able to answer these queries in milliseconds time. However, most of the time (depending on the multi-modal network) these local queries are not required. Thus, we are interested in better heuristics for locality filters, especially filters that could be computed implicitly during our approximate access-node preprocessing algorithm.

Multi-Criteria Search. In public transportation networks optimizing pure travel time might be unsatisfactory. Aspects like ticket fares, transfers and train classes demand for multi-criteria optimization. Thus, we are interested in multi-criteria shortest path algorithms in the context of multi-modal routing. Because with Access-Node Routing we are able to perform an independent algorithm on the public transportation

network, we are confident that it is possible to combine multi-criteria search on the public transportation network with single-criteria search on the 'outer' road network.

Appendix A

Data Structures

In this chapter of the appendix, we give implementation details on relevant data structures used throughout our experiments. These include the static graph used for most shortest path query algorithms. Since the static graph is static in the sense that it does not allow for insertion and deletion of edges, nodes and interpolation points, we present a further graph data structure which is dynamic and is primarily used to create graphs from the raw data and also in many parts of our contraction routines.

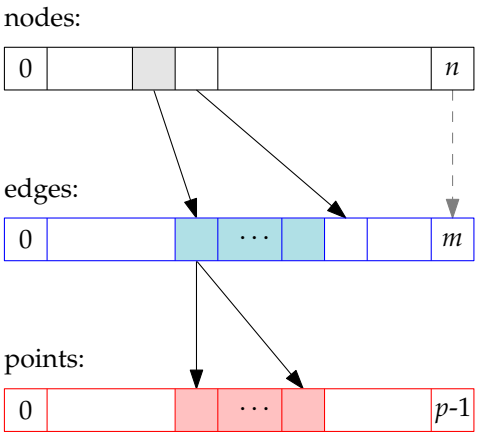
Our graphs are constructed to contain both time-independent and time-dependent edges at the same time. Hence, the travel time functions have to be stored in the graph data structure as well. In the static graph we use, roughly speaking, a fixed size vector to store all interpolation points (see later). The dynamic graph, however, contains objects of travel time functions supporting the link and merge-operations. We briefly discuss their assembly.

Multi-modal routing requires finite automata as input. We implemented a data structure which allows the computation of both forward and backward transitions, i.e., the computation of $\delta(q, \sigma)$ and $\delta^{-1}(q, \sigma)$ in constant time.

Finally, we briefly discuss the priority queue implementation used in our variants of the Dijkstra algorithm. We did not implement the priority queue by ourselves, but used an implementation of a binary heap from Schultes [Scho8b].

A.1. Graphs

We present two graph data structures. A static variant which is used throughout our query algorithms. It is mainly optimized regarding constant time iteration over the outgoing edges of a node as well as low memory consumption. The dynamic version of our graph data structure is mainly used during preprocessing. It allows




```
template<typename nodeDataT>
struct node {
  EdgeID firstEdge;
  nodeDataT data;
}
template<typename edgeDataT>
struct edge {
  NodeID targetNode;
  PointID firstPoint;
  PointID lastPoint;
  edgeDataT data;
}
struct point {
  unsigned int time;
  unsigned int weight;
}
```

Figure A.1.: Illustration of the basic setup of our static graph data structure.

node and edge insertions resp. deletions, whereas for edges we can create both time-independent and time-dependent edges in one graph.

A.1.1. Static Graphs

Our static graphs are based on a space efficient *forward-star representation* [CLRS01, MS08].

Nodes and Edges. When restricted to time-independent graphs, there are basically two arrays (we also refer to arrays as *vectors*): One representing the nodes and another representing the edges of the graph. Have a look at Figure A.1 and ignore the red vector at the bottom for the time being. Nodes and edges are referred to by their index in the respective array. We can interpret their (unique) indices as ids, hence, we address them by their `NodeID` resp. `EdgeID`.

While the nodes are ordered in arbitrary order, the edges are grouped by their tails. Hence, all edges (v, w) having the same tail v are next to each other. Furthermore, the outgoing edges of nodes with subsequent `NodeID`s are stored in subsequent order in the edge array. For that reason, it is sufficient to store *one* pointer per node v that points to the first outgoing edge of that node by its `EdgeID`. Iteration over the outgoing edges of a node v is, thus, possible by iterating over the section of the edge-vector enclosed by the `EdgeID`s (indices) v.`firstEdgeID` and $(v+1)$.`firstEdgeID` $-\,1$. The target `NodeID` is stored for each edge in the edge vector. To allow iteration over the node v having the highest `NodeID`, we insert a dummy-node at the end of the node vector that points toward a dummy-edge at the end of the edge vector.

To attach additional data to the nodes and edges, the entries of the node resp. edge vectors do not only contain pointers but `structs`. Besides the basic information that is necessary for every graph, the data section of the `struct` is defined through templates, thus, giving control over what data should be attached. For example, a multi-modal graph requires node and edge labels attached to the nodes and edges, while this is not required for an uni-modal graph. This allows us to tailor the graphs specifically to our requirements avoiding to waste memory unnecessarily.

Time-Dependency. In time-independent networks, each edge e is assigned a constant weight $w(e)$. While this information can be stored in the edge-`structs` easily, we extend the forward-star approach to edges for coping with time-dependency. Our edge functions are piecewise linear functions defined by a finite number of interpolation points. Thus, we introduce a third vector to store interpolation points. An interpolation point $\mathfrak{p}$ is addressed by its `PointID` (index in the points vector). Like edges for nodes, the interpolation points are grouped the same way with regard to the edges they belong to. Furthermore, they are sorted in ascending order regarding their value in the domain of the function (timestamp).

While evaluating the constant weight of an edge e given by its `EdgeID` can be done in constant time (we just need to access e's `struct` and read the weight), this is no longer true for time-dependent edges. To evaluate $f_e(\tau)$ for some departure time τ, we have to determine the nearest interpolation point $\mathfrak{p}_i$ of f_e 'in the future'. This requires some sort of search on the interpolation points. With binary search we, thus, imply a slow down in the size of $\mathcal{O}(\log(|f_e|))$ for evaluating $f_e(\tau)$ where $|f_e|$ denotes the number of interpolation points for f_e.

Mixing Edge Functions and Constant Weights. Most of our graphs contain both time-dependent and time-independent edges. However, the most part of the graphs is made up by the road network which is time-independent (see Section 3.2). On the other hand we have two (constant) weights per edge in the road network, foot and car. To save memory we, thus, modify the forward-star representation of the edges and interpolation points. Instead of only having one pointer per edge, we store two pointers: One to the first and one to the last index in the interpolation points array containing points belonging to the respective edge). Moreover, we introduce another flag `independent` which signals if the edge is time-independent. This can be implemented sparingly by using C++ bit fields. The first and last point pointers now have a context sensitive semantic.

- If the edge is time-dependent they contain the first and last `PointID` of interpolation points of the respective edge.

- On the other hand, if the edge is time-independent, they are used to store (up to) two constant weights. If the weight is not used, its value is set to ∞. Note that it is possible that there are edges in the road network which are open to either cars or pedestrians.

This 'misuse' of the point pointers allows us to store time-independent edges without the need of a special interpolation point indicating constant weights, nor additional variables in the edge `structs`. Because the most part of our graphs is time-independent (due to the huge road networks), the wasted space due to using two point pointers instead of one (like with edge pointers on nodes) is insignificant.

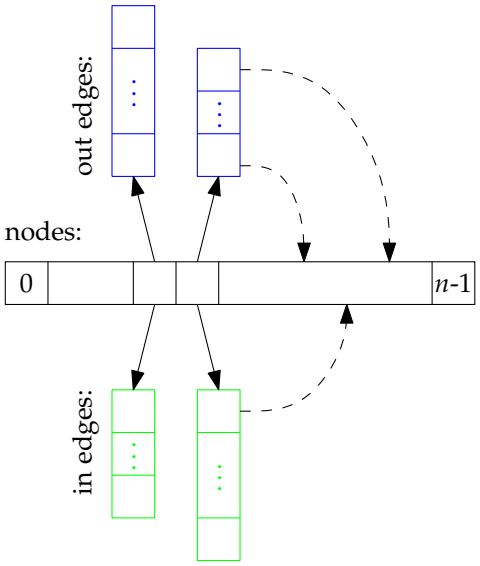
On the other hand this approach also has the downside of introducing special treatments to the edges at a number of places. An example is evaluating the constant weight of a road edge. First we have to check if the edge is time-independent. Second, if we request the car weight, we further have to check, if the value of `firstPointID` is set to ∞. Note that an ∞-weighted edge is equivalent to the edge simply not being contained in the graph. So, we had to perform extra work on an edge which should not exist in the first place. Though this problem can be solved by allowing multi-edges in the graph (in the case of a road being open for cars and pedestrians we would have two edges between the respective nodes), the memory overhead by that approach would outweigh the disadvantage of more complex checks.

A.1.2. Dynamic Graphs

Our dynamic graphs are based on the concept of incidence arrays. For each node we store its incident outgoing and incoming edges (ore more to the point: edge-tips). They are implemented as STL-vectors and, thus, are dynamic in the sense that they can grow and shrink during runtime. Further to the implementation of the data structure itself, we supply conversion routines to convert from dynamic to static graphs and vice versa. Look at Figure A.2 for an illustration of the data structure explained in the following.

Nodes and Edges. The nodes are composed of `structs` which are stored in a node vector. They are identified by their indices called `NodeID`. Note that deleting a node from the graph does not remove the respective entry from the vector, but just sets a flag indicating that the node has been removed. New nodes can be added by increasing the size of the node vector by one, thus, creating a new node-`struct` at the end of the array.

Regarding the edges, each node-`struct` has two dynamic edge vectors as members: One for outgoing and one for incoming edges-tips. The edge-`structs` contain a pointer




```
template<typename edgeDataT>
struct edge {
  NodeID targetNode;
  unsigned int w1, w2;
  PwLnFn f;
  edgeDataT data;
}
template<typename nodeDataT>
struct node {
  vector<edgeT> out;
  vector<inEdgeT> in;
  nodeDataT data;
}
struct in_edge {
  NodeID sourceNode;
}
```

Figure A.2.: Illustration of the basic setup of our dynamic graph data structure.

(a `NodeID` value) to their heads resp. tails. Let $e = (u, v)$ be an edge in the graph, then e is stored twice. Once in the outgoing edges vector of u and also in the incoming edges vector of v. Our graphs grow large, so storing all information twice for every edge can lead to a too high memory consumption. Thus, the attached data of the edges (weights and meta-data) is only stored once on the outgoing edge. So, in order to access for an incoming edge $e = (u, v)$ at v its weights or other data, we have to iterate over the outgoing edges vector of u until we find an edge with target node v. For that reason, our dynamic graphs do not support multi-edges, since multi-edges cannot be distinguished by their target resp. source nodes.

As with the static version of our graph data structure, meta data along nodes and edges is tailored by templates for maximum space efficiency.

Time-Dependency. When building a graph, especially time-dependent edges of it, it is important to have dynamic edge-weight functions (see next section). For that reason, the forward-edge-`structs` contain an object of our piecewise linear function structure. This allows the dynamic creation and removal of interpolation points. To cope with time-independent edges, additionally, two constant weights w_1 and w_2 are contained in the `structs`. Note that our trick sharing the memory for time-independent weights and pointers to the interpolation points like in the static graph is no longer possible with this data structure.

Vector-Compression. Our preprocessing routine for core-based routing creates and removes tremendous amounts of edges in the node-reduction step. We observed a constant increase in memory consumption during preprocessing, although the number of edges all together is not increasing by that much. For our biggest instances of multi-modal graphs this eventually led to running out of memory. This is due to the fact that STL-vectors are being implemented as dynamic arrays with geometric progression (see for example [CLRS01]).

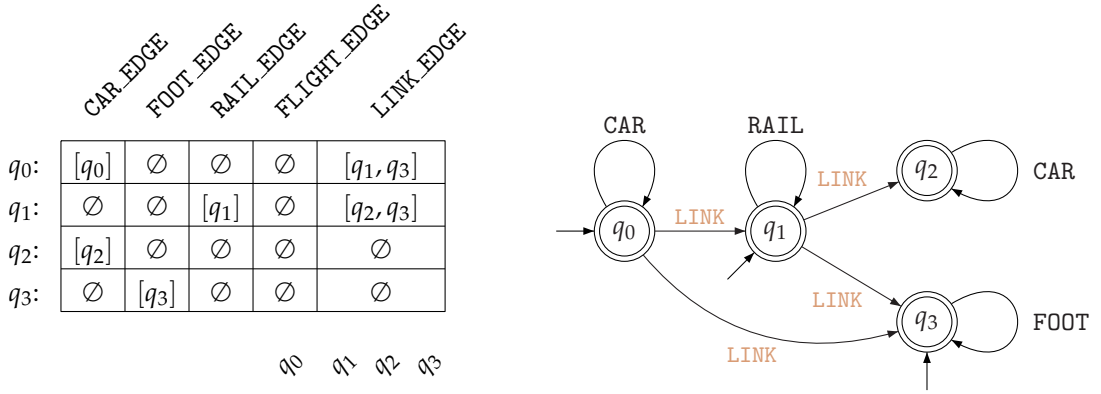
Thus, the size of the reserved memory is doubled if the array reaches its capacity limit. However, the size is only shrunk when the number of entries is below a certain threshold (usually 25–30%). Since this is unlikely to happen in our cases, arrays almost never shrink by themselves, leaving a lot of memory wasted. Thus, we implemented a vector-compression on the edges which is executed from time to time to resize the arrays exactly to fit their number of elements. While we lose guaranteed amortized complexity of $\mathcal{O}(1)$ for insertion and deletion by this approach, we do not run out of memory when performing a lot of edge insertions and deletions.

A.2. Piecewise Linear Functions

The piecewise linear functions used in our dynamic graph data structure are an implementation of the public transportation version of piecewise linear functions (cf. Section 3.1.2), thus, every segment of the function has a gradient of $\gamma = -1$. Basically a piecewise linear function consists of a dynamic vector of interpolation points $\mathfrak{p} = (\tau, f(\tau))$. Interpolation points are sorted in ascending order wrt. to the departure times τ in the vector and can be added and deleted by simply adding and removing entries into/from the vector. Note that adding an interpolation point may violate the FIFO-property on the function, thus, requiring some extra checks during insertions.

Our piecewise linear functions support both linking and merging. However, because our implementation is restricted to public transportation functions, we are not able to merge functions with constants (see Figure 4.1 on page 49). Note that linking a function to a constant or vice versa is unproblematic as the result is a 'pure' public transportation function with all segments having gradient -1.

Regarding profile queries on mixed networks utilizing both time-independent and time-dependent edges, not being able to use merging tuns out as a major drawback regarding the computation of profile queries since we cannot use the simple label correcting algorithm from [Dea99] (see also Section 4.2.2). For that reason, we have to evade to the multi label correcting approach (cf. Algorithm 4 on page 50) that manages without merge-operations, but with the penalty of significantly worse performance.



(a) The data structure. **(b)** Sample automaton.

Figure A.3.: This figure illustrates the NFA data structure (left) using the example of a small automaton (right).

A.3. Finite Automata

The finite automata used during our multi-modal queries are relatively small. Therefore, we do not need to pay that much attention to memory consumption. However, our multi-modal query algorithm (cf. Algorithm 5) requires us to determine the value of the function $\delta(q_i, \sigma_j)$ for some state q_i and label σ_j quickly.

Hence, when storing a finite automaton $\mathcal{A} = (\Sigma, Q, \delta, S, F)$ we enumerate Q from 0 to $|Q| - 1$ as `StateIDs`, Σ from 0 to $|\Sigma| - 1$ as `LabelIDs` and use a $|Q| \times |\Sigma|$ transition matrix T for storing the values of $\delta(q_i, \sigma_j)$. For the sake of comfort, the data structure of the matrix is a `multi_array` type from Boost (which is an STL extension library). The element at position $T[i, j]$ in the matrix contains a vector which contains the `StateIDs` that are members of the set $\delta(q_i, \sigma_j)$. Note that this vector can be empty, if $\delta(q_i, \sigma_j) = \emptyset$. This allows insertion and deletion of transitions by inserting and removing elements to/from their respective vectors. The query operation to determine the set of target states for some q_i and σ_j can, thus, be performed in constant time.

To mark states as initial resp. final states, there are two more bit-vectors of length $|Q| - 1$ that indicate whether the i'th state is an initial or final state. Also refer to Figure A.3 for an illustration of the data structure using a small automaton as example.

Inverse Automaton. For bi-directional routing we need the inverse automaton $\mathcal{A}^{-1}$. However, our data structure does not allow queries on the inverse transition function

```
% Number of states
4
% Initial states
S:0,1,3
% Final states
F:0,1,2,3
% Transitions (source state, label, target state)
T:0,CAR_EDGE,0
T:0,LINK_EDGE,1
T:0,LINK_EDGE,3
T:1,RAIL_EDGE,1
T:1,LINK_EDGE,2
T:1,LINK_EDGE,3
T:2,CAR_EDGE,2
T:3,FOOT_EDGE,3
```

Figure A.4.: Definition file of the finite automaton from Figure A.3.

δ^{-1} efficiently. Thus, when an inverse transition is requested for the first time from the automaton, we compute the inverse transition matrix T^{-1} from T. We do this by iterating through all fields in T and if a state k is contained in the transition vector of $T[i,j]$ we insert i into the inverse transition vector at position $T^{-1}[k,j]$. Note that this procedure has time complexity $\mathcal{O}(|Q|^2 \cdot |\Sigma|)$ which, however, constitutes no problem, since both Q and Σ are very small and the operation has to be performed only once, as we cache T^{-1} in memory for further queries.

Definition File. While our graphs are generated from raw data, the automata need to be defined manually. Thus, instead of using a binary file format, we use a human readable text format. The setup of a file defining the automaton of Figure A.3b is shown in Figure A.4. Lines starting with the `%` sign are comments. The first (uncommented) line contains the number of states. Each subsequent line has to start with one of the following letters: `S`,`F` or `T`, followed by a colon. Thereby `S`,`F` and `T` stand for **s**tart (initial) states, **f**inal states and **t**ransitions, respectively. The `S` and `T` lines then contain a comma separated list of initial resp. target state numbers (in the range of $0 \dots |Q| - 1$). Each `T`-line consists of one transition represented as a tuple (q, σ, q') with the semantic of $q' \in \delta(q, \sigma)$. These tuples are supplied as comma separated lists of three items, where the label σ is one of `CAR_EDGE`, `FOOT_EDGE`, `RAIL_EDGE`, `FLIGHT_EDGE` or `LINK_EDGE`.

A.4. Priority Queue

Our implementation of the priority queue is an adopted implementation by Schultes (see [Scho8b]). Elements of the priority queue are composed of `structs` that can be defined by templates. Since for some of our algorithms, we require the queue elements after they were deleted from the queue (this is especially true for our multi label correcting algorithm, cf. Algorithm 4), the elements are stored in a separate elements-vector and are identified by their index in the vector. The actual priority queue is implemented as a binary heap (with bottom-up heuristic) which only operates on the key and the index of the element in the elements-vector. This keeps the heap-vector small while still being able to access elements that were once deleted from the queue. However, the penalty of this approach is that we have to keep track of the queue elements manually in our algorithms which results in some additional memory consumption.

Appendix B

Raw Data Processing

In this chapter we discuss the processing of raw data from which our graphs are created. This involves the creation of the road network, the railway network and the flight network. The multi-modal graphs are not created from raw data directly. Instead, they are created by merging several uni-modal graphs (see Section 3.5 for details). While regarding both the road and railway networks, we had access to 'real' raw data, the data for our flight networks is based on timetables of two major flight alliances that were publicly available on the Internet at the time of writing.

B.1. Road Data

The road data was kindly given to us by PTV AG [ptv79]. It contains two sets, Europe and North America (USA and Canada), both from the year 2006 in the *Advanced Geographic Format* (AGF). Each of the sets is composed of a number of subdirectories. Each directory contains the data of one country (Europe) resp. one state (America). Each directory then consists of a set of text files that contain information about nodes (.NDF), links (.LNK) and link directions (.LKD) among other files that are not relevant for our application. We omit a detailed description of the file structures at this point.

Relevant information we extract besides the graph structure itself, includes: Geographical coordinates (x and y) on the nodes, the average speed as well as the geographical length of the links and whether the links are open for pedestrians, cars or both. The links file refers to the nodes by their id numbers, which are defined in the respective node file. Unfortunately, these id numbers are not globally unique, i.e., nodes belonging to different countries at different geographical locations may well have the same id numbers. For that reason, we preprocess the data by unifying the `.NDF`, `.LNK` and `.LKD` files from each subdirectory. Thereby, two nodes are unified if their ge-

```
*Z 27988 RD____    01                                  % 27988 RD____    01
*G DNR 8000247 5400004                                 % 27988 RD____    01
*A VE 8000247 5400004 000000                           % 27988 RD____    01
*A FB 8000247 5400004                                  % 27988 RD____    01
*A RD 8000247 5400004                                  % 27988 RD____    01
*A GR 8005352 5400004                                  % 27988 RD____    01
*GR 8005353 8005352 5400004 Cheb(Gr)                   % 27988 RD____    01
8000247 Marktredwitz              2112                 % 27988 RD____    01
8000613 Arzberg(Oberfr)      2119 2120                 % 27988 RD____    01
8005352 Schirnding           2124 2126                 % 27988 RD____    01
5400004 Cheb                 2140                      % 27988 RD____    01
```

Figure B.1.: Sample set of raw data belonging to a local train from Marktredwitz to Cheb at the German-Czech border.

ographical coordinates match. This is required in order to have connections across country borders. Note that by these means we might lose some detail in the network, e.g., small roundabouts consisting of many nodes so close together that they cannot be discriminated by their coordinates. However, these effects are rare and therefore negligible.

The unified data is then parsed directly generating the graph representing our road network model (cf. Section 3.2). Thereby, the average travel speed is converted into average travel time by using the link length. Furthermore, for links that are open for pedestrians we assume an average travel speed of $4\,\mathrm{m/s}$. Note that ferry links are included in the raw data by the means of travel time. However, these links are time-independent, i.e., the travel time is valid for all times of day.

B.2. Railway Data

The railway data is based on different timetables from the German Railway company and was kindly given to us by HaCon [hac84]. For our graphs we use two timetables. The timetable of the winter period 1996/1997 containing most long range trains of Europe and the timetable of the winter period 2000/2001 containing the German train network together with a few local public transportation networks.

The raw data consists of a set of text files for each timetable. There is one file defining stations together with their id numbers, names and geographical locations. Another file contains the transfer time for each station. The train connections are modeled as movements of trains. Figure B.1 shows a sample train as it occurs in the raw data. The first lines (those beginning with a star) contain meta information about the train which is not interesting to us. Next, each line consists of one station the

From ATHENS, GREECE (ATH)

TO: BUDAPEST, HUNGARY (BUD)

From - To Validity		Days	Dep	Arr	Flight	Aircraft	Elapsed time
-	25Oct	123456	**05:20**	**06:20**	MA231	736	02:00
27Oct	-	1 5	**05:20**	**06:25**	MA231	F70	02:05
-	25Oct	2 67	**15:40**	**16:40**	MA233	73G	02:00
26Oct	-	2 5 7	**15:40**	**16:45**	MA233	73G	02:05
-	24Oct	1 345	**16:30**	**17:30**	MA233	73G	02:00
27Oct	-	1 34	**16:30**	**17:35**	MA233	73G	02:05

To ATHENS, GREECE (ATH)

FROM: BUDAPEST, HUNGARY (BUD)

From - To Validity		Days	Dep	Arr	Flight	Aircraft	Elapsed time
-	25Oct	2 67	**11:55**	**14:55**	MA232	73G	02:00
26Oct	-	2 5 7	**12:00**	**15:00**	MA232	73G	02:00
-	30Oct	1 345	**12:50**	**15:50**	MA232	73G	02:00
-	26Oct	12345 7	**23:35**	**02:35+1**	MA230	736	02:00
30Oct	-	4	**23:35**	**02:35+1**	MA230	736	02:00

Figure B.2.: Excerpt from the original pdf-timetable of the Oneworld [One99] flight alliance between Athens (ATH), Greece and Budapest (BUD), Hungary.

train is passing through and contains the station id, its name, the arrival time and the departure time at that station.

In a first step we convert the raw data into an intermediate format representing the mathematical definition of a timetable as introduced in Section 3.3.1 on page 18 more closely. Hence, we create one file containing station definitions (for each station a newly assigned and unique id number, the name, the geographical location and the transfer time) as well as a file of elementary connections. Each line of this file consists of the departure station id, the arrival station id, a newly created train id and the departure resp. arrival times. Because we are interested in creating a graph wrt. the realistic time-dependent model which involves the concept of routes (cf. Section 3.3.4), we group the elementary connections by trains like in the raw data. This makes it easier to create equivalent trains during the creation of the model.

In a second step we create the graph of the realistic time-dependent model with constant transfer times from the intermediate format. To identify equivalent trains we parse the connections file line by line creating train objects along the way. The trains are then sorted lexicographically (regarding the station IDs), thus, allowing equivalent trains to be merged into routes easily. These are then be used for creating the nodes and edges of the final graph. The edge weights are set according to the definition of the realistic time-dependent model. Along the connection edges we insert an interpolation point for each elementary connection from the intermediate file belonging to a train using the route the respective edge belongs to.

B.3. Flight Data

Regarding the flight networks we did not have raw timetable data at our disposal. Thus, we decided creating the flight data from public timetables available on the Internet. We used the timetables provided by two major flight alliances: StarAlliance [Sta97] and Oneworld [One99]. The data format in which the timetables are provided by both

alliances is the portable document format (pdf). Figure B.2 shows a sample of the timetable of the Oneworld flight alliance.

In a first step the pdf-files are dumped into a human readable text file format. From these text files we then create an intermediate format consisting of elementary connections resembling the formal definition of timetables as introduced in Section 3.3.1 on page 18. Each elementary connection in the intermediate format consists of four values: The source resp. target airport IATA-codes (see [Int45]) and the departure resp. arrival times of the specific flight.

Unfortunately, this data is not sufficient to create the flight model graphs as introduced in Section 3.4. First of all, the departure/arrival times are given wrt. the local timezones where the respective airports are. Second, the timetable does not contain geographical information which is, however, mandatory to us in order to link the flight network to the other networks (cf. Section 3.5). For that reason, for each IATA-code contained in the intermediate files, we query a website [Gre96] that returns a record containing geographical coordinates, the airport's full name and the local time zone. This information is written into a separate text file.

Finally, the flight model graph is created on the basis of the two intermediate files (the file containing airport information plus the file containing elementary connections). Our multi-modal graphs use the flight class model introduced in Section 3.4.3. We thus consider two flights as equivalent if they belong to the same flight alliance. In our case these are exactly Star Alliance and Oneworld. The different transfer times $\mathcal{T}^{\text{check-in}}$, $\mathcal{T}^{\text{check-out}}$ and $\mathcal{T}^{\text{transfer}}$ are synthetic, since we were not able to acquire this data. We set these values to 120 minutes, 60 minutes and 90 minutes, respectively for each flight class and every airport. Transfers between different flight alliances are set to 150 minutes.

Acknowledgments

First of all, I would like to thank Prof. Dr. Dorothea Wagner for giving me the opportunity to work on this highly interesting topic; Daniel Delling and Martin Holzer for giving me support and the plenty of time we spent having discussions which were always very inspiring. They were always available and helpful when I had a question or problem.

I spent three months working on this thesis at the University of Patras in Greece. I would really like to thank Prof. Dr. Christos Zaroliagis for welcoming me in his research group. The very friendly and relaxed working environment made it a pleasant stay. Furthermore, I would like to thank Georgia Nikolopoulou who arranged for my arrival and accommodation which made my start in Greece a lot easier.

Finally, I would like to thank my parents who have supported me through my whole studies and without whom I would not have had the chance to write this thesis.

Last but not least, I'd like to thank all the people not mentioned by name who inspired me or distracted me when I needed a break. In particular, my closest friends on whom I could always rely and who were supporting me throughout the whole time. *Thanks!*

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig angefertigt habe und nur die angegebenen Hilfsmittel und Quellen verwendet wurden.

Karlsruhe, den 30. März 2009

Bibliography

[BBH+08] Chris Barrett, Keith Bisset, Martin Holzer, Goran Konjevod, Madhav V. Marathe, and Dorothea Wagner. Engineering Label-Constrained Shortest-Path Algorithms. In *Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management (AAIM'08)*, volume 5034 of *Lecture Notes in Computer Science*, pages 27–37. Springer, June 2008.
(Cited on page 53.)

[BBH+09] Chris Barrett, Keith Bisset, Martin Holzer, Goran Konjevod, Madhav V. Marathe, and Dorothea Wagner. Engineering Label-Constrained Shortest-Path Algorithms. In Demetrescu et al. [DGJ09]. To appear.
(Cited on pages 4 and 136.)

[BBJ+02] Chris Barrett, Keith Bisset, Riko Jacob, Goran Konjevod, and Madhav V. Marathe. Classical and contemporary shortest path problems in road networks: Implementation and experimental analysis of the transims router. In *ESA '02: Proceedings of the 10th Annual European Symposium on Algorithms*, pages 126–138, London, UK, 2002. Springer-Verlag.
(Cited on page 4.)

[BD08] Reinhard Bauer and Daniel Delling. SHARC: Fast and Robust Unidirectional Routing. In Ian Munro and Dorothea Wagner, editors, *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX'08)*, pages 13–26. SIAM, April 2008.
(Cited on pages 76 and 96.)

[BD09] Reinhard Bauer and Daniel Delling. SHARC: Fast and Robust Unidirectional Routing. *ACM Journal of Experimental Algorithmics*, 2009. Special Section devoted to selected best papers presented at ALENEX'08. To ap-

pear.
(Cited on pages 3 and 96.)

[BDS+08] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. In Catherine C. McGeoch, editor, *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 303–318. Springer, June 2008.
(Cited on pages 3, 62, and 76.)

[BDS+09] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. Invited to a special issue of the ACM Journal of Experimental Algorithmics devoted to the best papers of WEA 2008, 2009.
(Cited on pages 3, 76, and 79.)

[BDW07] Reinhard Bauer, Daniel Delling, and Dorothea Wagner. Experimental Study on Speed-Up Techniques for Timetable Information Systems. In Christian Liebchen, Ravindra K. Ahuja, and Juan A. Mesa, editors, *Proceedings of the 7th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'07)*, pages 209–225. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.
(Cited on pages 2, 4, 64, and 69.)

[BDW09] Reinhard Bauer, Daniel Delling, and Dorothea Wagner. Experimental Study on Speed-Up Techniques for Timetable Information Systems. *Networks*, 2009. Accepted for publication, to appear.
(Cited on page 4.)

[Bel58] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.
(Cited on pages 2 and 47.)

[Ben75] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.
(Cited on page 34.)

[BFM+07] Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes. In Transit to Constant Shortest-Path Queries in Road

Networks. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, pages 46–59. SIAM, 2007.
(Cited on pages 3, 62, and 96.)

[BFM09] Holger Bast, Stefan Funke, and Domagoj Matijevic. TRANSIT Ultrafast Shortest-Path Queries with Linear-Time Preprocessing. In Demetrescu et al. [DGJ09]. Accepted for publication, to appear.
(Cited on page 3.)

[BFSS07] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. Fast Routing in Road Networks with Transit Nodes. *Science*, 316(5824):566, 2007.
(Cited on pages 3, 62, and 96.)

[BJ04] Gerth Brodal and Riko Jacob. Time-dependent Networks as Models to Achieve Fast Exact Time-table Queries. In *Proceedings of ATMOS Workshop 2003*, pages 3–15, 2004.
(Cited on page 45.)

[BJM00] Chris Barrett, Riko Jacob, and Madhav Marathe. Formal-language-constrained path problems. *SIAM J. Comput.*, 30(3):809–837, 2000.
(Cited on pages 4, 52, 53, and 136.)

[CH66] K. Cooke and E. Halsey. The Shortest Route Through a Network with Time-Dependent Intermodal Transit Times. *Journal of Mathematical Analysis and Applications*, (14):493–498, 1966.
(Cited on page 41.)

[CH08] Horst Czichos and Manfred Hennecke, editors. *HÜTTE – Das Ingenieurwissen*. Springer, Berlin, Heidelberg, 33 edition, 2008.
(Not cited.)

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.
(Cited on pages 140 and 144.)

[Dan62] George B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1962.
(Cited on pages 2, 4, and 63.)

[Dea99] Brian C. Dean. Continuous-Time Dynamic Shortest Path Algorithms. Master's thesis, Massachusetts Institute of Technology, 1999.
(Cited on pages 41, 48, 51, 101, and 144.)

[Del08] Daniel Delling. Time-Dependent SHARC-Routing. In *Proceedings of the 16th Annual European Symposium on Algorithms (ESA'08)*, volume 5193 of *Lecture Notes in Computer Science*, pages 332–343. Springer, September 2008. Best Student Paper Award - ESA Track B.
(Cited on pages 72, 76, and 80.)

[Del09a] Daniel Delling. *Engineering and Augmenting Route Planning Algorithms*. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2009.
(Cited on pages 2, 16, 61, 82, 92, 95, 107, 123, and 125.)

[Del09b] Daniel Delling. Time-Dependent SHARC-Routing. *Algorithmica*, 2009. Special section devoted to selected best papers of ESA'08. to appear.
(Cited on pages 3 and 16.)

[DGJ09] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors. *Shortest Paths: Ninth DIMACS Implementation Challenge*. DIMACS Book. American Mathematical Society, 2009. To appear.
(Cited on pages 2, 155, 157, 158, 159, 160, and 165.)

[DHM+09] Daniel Delling, Martin Holzer, Kirill Müller, Frank Schulz, and Dorothea Wagner. High-Performance Multi-Level Routing. In Demetrescu et al. [DGJ09]. Accepted for publication, to appear.
(Cited on page 3.)

[Dij59] Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.
(Cited on pages 2, 4, 41, 44, 46, and 61.)

[DMS08] Yann Disser, Matthias Müller–Hannemann, and Mathias Schnee. Multi-Criteria Shortest Paths in Time-Dependent Train Networks. In Catherine C. McGeoch, editor, *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 347–361. Springer, June 2008.
(Cited on pages 4 and 42.)

[DN08] Daniel Delling and Giacomo Nannicini. Bidirectional Core-Based Routing in Dynamic Time-Dependent Road Networks. In Seok-Hee Hong, Hiroshi Nagamochi, and Takuro Fukunaga, editors, *Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC'08)*, volume 5369 of *Lecture Notes in Computer Science*, pages 813–824. Springer, December 2008.
(Cited on pages 3, 80, 91, and 92.)

[DPW08] Daniel Delling, Thomas Pajor, and Dorothea Wagner. Engineering Time-Expanded Graphs for Faster Timetable Information. In *Proceedings of the 8th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'08)*, Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, September 2008.
(Cited on pages 3, 4, 21, and 28.)

[DSSW06] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Highway Hierarchies Star. In Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors, *9th DIMACS Implementation Challenge - Shortest Paths*, November 2006.
(Cited on pages 68 and 92.)

[DSSW09a] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering Route Planning Algorithms. In Jürgen Lerner, Dorothea Wagner, and Katharina A. Zweig, editors, *Algorithmics of Large and Complex Networks*, Lecture Notes in Computer Science. Springer, 2009. To appear.
(Cited on pages 2, 14, and 61.)

[DSSW09b] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Highway Hierarchies Star. In Demetrescu et al. [DGJ09]. Accepted for publication, to appear.
(Cited on page 3.)

[DW07] Daniel Delling and Dorothea Wagner. Landmark-Based Routing in Dynamic Graphs. In Camil Demetrescu, editor, *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*, pages 52–65. Springer, June 2007.
(Cited on page 2.)

[Gei08] Robert Geisberger. Contraction Hierarchies. Master's thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2008.
(Cited on pages 3, 76, and 96.)

[GH05] Andrew V. Goldberg and Chris Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 156–165, 2005.
(Cited on pages 2, 4, 63, 66, 68, and 69.)

[GKW06a] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. In *Proceedings of*

the 8th Workshop on Algorithm Engineering and Experiments (ALENEX'06), pages 129–143. SIAM, 2006.
(Cited on page 3.)

[GKW06b] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Reach for A*: Shortest Path Algorithms with Preprocessing. In Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors, *9th DIMACS Implementation Challenge - Shortest Paths*, November 2006.
(Cited on page 3.)

[GKW07] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Better Landmarks Within Reach. In Camil Demetrescu, editor, *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*, pages 38–51. Springer, June 2007.
(Cited on page 92.)

[GKW09] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Reach for A*: Shortest Path Algorithms with Preprocessing. In Demetrescu et al. [DGJ09]. Accepted for publication, to appear.
(Cited on page 3.)

[Gre96] Great Circle Mapper. `http://gc.kls2.com`, 1996.
(Cited on page 152.)

[GSSD08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In Catherine C. McGeoch, editor, *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, June 2008.
(Cited on pages 3, 76, 84, and 96.)

[GW05] Andrew V. Goldberg and Renato F. Werneck. Computing Point-to-Point Shortest Paths from External Memory. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05)*, pages 26–40. SIAM, 2005.
(Cited on pages 2, 66, 68, and 69.)

[hac84] HaCon - Ingenieurgesellschaft mbH, 1984. `http://www.hacon.de`.
(Cited on page 150.)

[HKMS06] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In Camil

Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors, *9th DIMACS Implementation Challenge - Shortest Paths*, November 2006.
(Cited on pages 70 and 71.)

[HNR68] Peter E. Hart, Nils Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.
(Cited on pages 2, 4, and 66.)

[Holo8] Martin Holzer. *Engineering Planar-Separator and Shortest-Path Algorithms*. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2008.
(Cited on pages 4, 53, and 136.)

[HSW04] Martin Holzer, Frank Schulz, and Thomas Willhalm. Combining Speed-up Techniques for Shortest-Path Computations. In *Proceedings of the 3rd Workshop on Experimental Algorithms (WEA'04)*, volume 3059 of *Lecture Notes in Computer Science*, pages 269–284. Springer, 2004.
(Cited on page 3.)

[HSWW06] Martin Holzer, Frank Schulz, Dorothea Wagner, and Thomas Willhalm. Combining Speed-up Techniques for Shortest-Path Computations. *ACM Journal of Experimental Algorithmics*, 10, 2006.
(Cited on page 3.)

[Int45] International Air Transport Association. `http://www.iata.org`, 1945.
(Cited on page 152.)

[Jr.56] L.R. Ford Jr. Network flow theory. Paper P-923, The RAND Corperation, Santa Moncia, California, August 1956.
(Cited on pages 2 and 47.)

[Karo7] George Karypis. METIS - Family of Multilevel Partitioning Algorithms, 2007.
(Cited on pages 3 and 70.)

[Keno4] Matthew B. Kennel. Kdtree 2: Fortran 95 and c++ software to efficiently search for near neighbors in a multi-dimensional euclidean space. 2004.
(Cited on pages 35 and 38.)

[KK98] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998.
(Cited on page 70.)

[Kle] S. Kleene. *Representation of Events in Nerve Nets and Finite Automata*, pages 3–42. Princeton University Press, Princeton, N.J.
(Cited on page 12.)

[KMS05] Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Acceleration of Shortest Path and Constrained Shortest Path Computation. In *Proceedings of the 4th Workshop on Experimental Algorithms (WEA'05)*, Lecture Notes in Computer Science, pages 126–138. Springer, 2005.
(Cited on page 3.)

[Lau97] Ulrich Lauther. Slow Preprocessing of Graphs for Extremely Fast Shortest Path Calculations, 1997. Lecture at the Workshop on Computational Integer Programming at ZIB.
(Cited on page 69.)

[Lau04] Ulrich Lauther. An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. In *Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung*, volume 22, pages 219–230. IfGI prints, 2004.
(Cited on pages 3, 69, and 71.)

[Moo91] Andrew Moore. An introductory tutorial on kd-trees. Technical Report Technical Report No. 209, Computer Laboratory, University of Cambridge, Pittsburgh, PA, 1991.
(Cited on page 35.)

[Mor92] H. Moritz. Geodetic reference system 1980. *Journal of Geodesy*, 66(2):187–192, June 1992.
(Cited on page 38.)

[MS04] Burkhard Monien and Stefan Schamberger. Graph Partitioning with the Party Library: Helpful-Sets in Practice. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'04)*, pages 198–205. IEEE Computer Society, 2004.
(Cited on pages 3 and 70.)

[MS07] Matthias Müller–Hannemann and Mathias Schnee. Finding All Attractive Train Connections by Multi-Criteria Pareto Search. In *Algorithmic Methods for Railway Optimization*, volume 4359 of *Lecture Notes in Computer Science*, pages 246–263. Springer, 2007.
(Cited on pages 4 and 42.)

[MS08] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008.
(Cited on page 140.)

[MSS+06] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning Graphs to Speedup Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 11:2.8, 2006.
(Cited on pages 69 and 70.)

[MSWZ07] Matthias Müller–Hannemann, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Timetable Information: Models and Algorithms. In *Algorithmic Methods for Railway Optimization*, volume 4359 of *Lecture Notes in Computer Science*, pages 67–90. Springer, 2007.
(Cited on page 3.)

[Mül06] Kirill Müller. Design and Implementation of an Efficient Hierarchical Speed-up Technique for Computation of Exact Shortest Paths in Graphs. Master's thesis, Universität Karlsruhe (TH), Fakultät für Informatik, June 2006.
(Cited on page 3.)

[MW01] Matthias Müller–Hannemann and Karsten Weihe. Pareto Shortest Paths is Often Feasible in Practice. In *Proceedings of the 5th International Workshop on Algorithm Engineering (WAE'01)*, volume 2141 of *Lecture Notes in Computer Science*, pages 185–197. Springer, 2001.
(Cited on pages 4 and 42.)

[NDLS08] Giacomo Nannicini, Daniel Delling, Leo Liberti, and Dominik Schultes. Bidirectional A* Search for Time-Dependent Fast Paths. In Catherine C. McGeoch, editor, *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 334–346. Springer, June 2008.
(Cited on pages 2, 65, 68, and 95.)

[One99] Oneworld Management Company Ltd. `http://www.oneworld.com`, 1999.
(Cited on pages 118 and 151.)

[OR90] Ariel Orda and Raphael Rom. Shortest-Path and Minimum Delay Algorithms in Networks with Time-Dependent Edge-Length. *Journal of the ACM*, 37(3):607–625, 1990.
(Cited on page 25.)

[Pel07] Francois Pellegrini. SCOTCH: Static Mapping, Graph, Mesh and Hypergraph Partitioning, and Parallel and Sequential Sparse Matrix Ordering Package, 2007.
(Cited on pages 3 and 70.)

[PSWZ04] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Experimental Comparison of Shortest Path Approaches for Timetable Information. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX'04)*, pages 88–99. SIAM, 2004.
(Cited on page 3.)

[PSWZ07] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient Models for Timetable Information in Public Transportation Systems. *ACM Journal of Experimental Algorithmics*, 12:Article 2.4, 2007.
(Cited on pages 3, 4, 24, 26, 43, and 45.)

[ptv79] PTV AG - Planung Transport Verkehr, 1979. `http://www.ptv.de`.
(Cited on page 149.)

[RS59] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3:114–125, 1959.
(Cited on page 12.)

[Sch05] Frank Schulz. *Timetable Information and Shortest Paths*. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2005.
(Cited on pages 3 and 19.)

[Sch08a] Dennis Schieferdecker. Systematic Combination of Speed-Up Techniques for exact Shortest-Path Queries. Master's thesis, Universität Karlsruhe (TH), Fakultät für Informatik, January 2008.
(Cited on pages 2, 3, 76, 82, 91, 94, and 96.)

[Sch08b] Dominik Schultes. *Route Planning in Road Networks*. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, February 2008.
(Cited on pages 139 and 147.)

[SS05] Peter Sanders and Dominik Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05)*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2005.
(Cited on pages 3 and 76.)

[SS06a] Peter Sanders and Dominik Schultes. Engineering Highway Hierarchies. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA'06)*, volume 4168 of *Lecture Notes in Computer Science*, pages 804–816. Springer, 2006.
(Cited on pages 3 and 76.)

[SS06b] Peter Sanders and Dominik Schultes. Robust, Almost Constant Time Shortest-Path Queries in Road Networks. In Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors, *9th DIMACS Implementation Challenge - Shortest Paths*, November 2006.
(Cited on pages 3, 96, and 106.)

[SS09] Peter Sanders and Dominik Schultes. Robust, Almost Constant Time Shortest-Path Queries in Road Networks. In Demetrescu et al. [DGJ09]. Accepted for publication, to appear.
(Cited on page 3.)

[Sta97] Star Alliance. `http://www.staralliance.com`, 1997.
(Cited on pages 118 and 151.)

[SV86] Robert Sedgewick and Jeffrey S. Vitter. Shortest Paths in Euclidean Graphs. *Algorithmica*, 1(1):31–48, 1986.
(Cited on page 4.)

[SWW99] Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. In *Proceedings of the 3rd International Workshop on Algorithm Engineering (WAE'99)*, volume 1668 of *Lecture Notes in Computer Science*, pages 110–123. Springer, 1999.
(Cited on pages 2 and 3.)

[WW03] Dorothea Wagner and Thomas Willhalm. Geometric Speed-Up Techniques for Finding Shortest Paths in Large Sparse Graphs. In *Proceedings of the 11th Annual European Symposium on Algorithms (ESA'03)*, volume 2832 of *Lecture Notes in Computer Science*, pages 776–787. Springer, 2003.
(Cited on page 2.)

[WWZ05] Dorothea Wagner, Thomas Willhalm, and Christos Zaroliagis. Geometric Containers for Efficient Shortest-Path Computation. *ACM Journal of Experimental Algorithmics*, 10:1.3, 2005.
(Cited on page 2.)