Project Number 288094

eCOMPASS

eCO-friendly urban Multi-modal route PlANning Services for mobile uSers

STREP
Funded by EC, INFSO-G4(ICT for Transport) under FP7

eCOMPASS – TR – 020

Multimodal Profile Queries

Andreas Bauer

June 2013

Karlsruhe Institute of Technology

Multimodal Profile Queries

Bachelor Thesis of

Andreas Bauer

At the Department of Informatics
Institute of Theoretical Informatics

Reviewer:	Prof. Dr. Dorothea Wagner
Second reviewer:	Prof. Dr. Peter Sanders
Advisor:	Dipl.-Inform. Julian Dibbelt
Second advisor:	Dipl.-Inform. Thomas Pajor

Duration: 10 January 2012 – 9 May 2012

Selbstständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht habe und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Karlsruhe, 9.5.2012

Abstract

Route planning has many applications in the real world. We can use it for computing the best journey for both road or public transportation networks. In the latter, the journeys are restricted by a schedule. The traveltime then depends on the departure time. To find the shortest traveltime for a given departure time is called a *time query*, to find the shortest traveltime for all departure times is called a *profile query*. In this thesis, we develop and evaluate algorithms that perform such profile queries for a *multimodal network*. A multimodal network is a network that consists of several modes of transports.
Some sequences of modes of transports are undesirable. For example, a user might not want to take a taxi after exiting a train. This can be modeled by assigning labels to all edges of the subnetwork, and only to consider those paths as valid where the labels of the edges produce a word of a formal language.
We develop two algorithms, named the *Function* and the *Label Algorithm*. We present improvements to both algorithms and evaluate them on real world data. We achieve a speedup of up to one order of magnitude to a hypothetical algorithm that knows each relevant departure time in advance and performs a multimodal, time-dependent generalization of Dijkstra's Algorithm for each of them.

Deutsche Zusammenfassung

Routenplanung hat viele Anwendungen in der realen Welt. Wir können sowohl für Straßennetze als auch für öffentliche Verkehrsnetze schnellste Reisewege finden. Bei letzterem müssen wir einen Zeitplan beachten. Die Reisezeit ist dann vom Abfahrtszeitpunkt abhängig. Die kürzeste Reisezeit für einen gegebenen Abfahrtszeitpunkt zu finden ist eine *Zeitanfrage*, sie für alle Abfahrtszeitpunkte zu finden ist eine *Profilsuche*. In dieser Arbeit entwickeln und evaluieren wir Algorithmen, die solche Profilsuchen auf *multimodalen Netzen* ausführen. Ein multimodales Netz ist ein Netz, das verschiedene Transportarten besitzt.
Einige Abfolgen der Transportarten sind unerwünscht. So könnte es sein, dass der Benutzer kein Taxi nehmen will, nachdem er einen Zug verlassen hat. Dies kann dadurch modelliert werden, dass jeder Kante im Graphen Label zugeordnet werden. Nur Pfade, bei denen die Label der Kanten ein Wort aus einer formalen Sprache ergeben, werden als gültig angesehen.
Wir entwickeln zwei Algorithmen, den *Function* und den *Label Algorithmus*. Wir entwickeln Verbesserungen für beide Algorithmen und testen sie auf realen Daten. Wir erreichen eine Beschleunigung von bis zu einer Größenordnung gegenüber einem hypothetischen Algorithmus, der alle relevanten Abfahrtszeitpunkte bereits kennt und für jeden dieser Zeitpunkte eine multimodale, zeitabhängige Erweiterung von Dijkstra's Algorithmus ausführt.

Contents

1. Introduction

Today it is natural to use technical aids to plan a journey and not just rely on normal maps. Navigation systems for cars are common, and more and more people do not inform themselves about rail routes at the counter, but on the Internet. As a matter of fact, route planning is "one of the showpieces of real-world applications of algorithmics" [DSSW09].

The basic approach for route planning is to first model the real world data as a graph and then to search a shortest path on it. This can be done with Dijkstra's Algorithm [Dij59], or one of its many extensions developed in the last years. For road networks, research in this field lead to ever faster algorithms. Even on networks like the European road network queries can be answered in milliseconds [DSSW09], and some techniques not based on Dijkstra's Algorithm only need microseconds [ADGW10]. Networks derived from timetable information, like rail networks, have proven to be structurally different from road networks [BDW07, Bas09]. In most research, these networks are also modeled as graphs (albeit there are exceptions [DPW12a]). How to generate graphs from timetables efficiently was subject of much consideration. In [PSWZ08], Pyrga et al. present the time-expanded and time-dependent models. Though not part of timetable information, they also incorporate transfer times when switching between trains into both models, which is important for feasibility in real world applications. Evaluations there and in [BDW07] indicate that the time-dependent model is the most promising approach for modeling rail networks.

All these algorithms are, however, specialized on one type of network respectively. Multimodal route planning, where different kinds of transports may be used, is not possible using only the aforementioned algorithms. On the one hand, we need to find a combined model for all the networks. On the other hand, using the networks in any order is not always feasible in practice [DPW09]: A person can get to the train station by car, but at the destination it will be no longer available for him when they enter the road network again.

Modeling such multimodal networks is subject of [Paj09]. The idea is to model each network independently as a graph. For each of these graphs, a unique label is introduced and assigned to each edge. After that, the networks are linked. For a path, the labels of the edges form a word, and only when the word is from a given formal language the path is considered valid. To search a shortest path under this constraint is called the Label Constraint Shortest Path Problem. A formal definition of this problem and an extension of Dijkstra's Algorithm to solve it efficiently for regular languages are shown in [BJM00].

There are several improvements to this approach. Query times are improved by access nodes [DPW09], goal directed techniques are applied [KLPC11, KLC12], and the search on the road part is accelerated by using Contraction Hierarchies [DPW12b].

Most of these publications focus on computing the earliest arrival time for a fixed departure time. Such a query is called a time query. However, the traveltime is not the only relevant optimization criterion. The arrival and departure times themselves are important, too. A user has to consider whether a considerably shorter traveltime justifies an earlier departure or a later arrival time. It is hard to define how important each of these criterions are, even for the user himself. It is therefore useful not to compute only the traveltime for one departure time, but for all departure times. This is called a profile query. The result is presented as a function that maps each departure time to its shortest traveltime.

In [Dea99], Dean presents an algorithm for computing profile queries on unimodal networks. The algorithm has almost the same structure as Dijkstra's Algorithm. But instead of scalar values, functions are stored at the nodes as tentative distances and are used as edge weights.

In [DKP10] Delling et. al. improve the time-dependent model of the railway network, resulting in a model with less nodes, the color model. The authors furthermore take a different approach to compute profile queries on this network. They store tentative arrival times for all possible departure times from the source station. These arrival times are then settled successively as in Dijkstra's algorithm. Unfortunately, we do not have a practical bound for possible departures from the source, as we also use road networks where one can start at any time.

Our Contribution. In this work we adapt the algorithm presented in [Dea99] for multimodal networks. We call this the Function Algorithm. We consider drawbacks of this approach, and develop another algorithm, which we call the Label Algorithm. We achieve a speedup up to an order of magnitude over a hypothetical algorithm that knows each relevant departure time in advance and performs a multimodal, time-dependent version of Dijkstra's Algorithm for each of them.

Overview. In Chapter 2, we explain the foundations on which the algorithms are based, as well as the models. In Chapter 3, we present the Function and Label Algorithm. Both algorithms are evaluated in Chapter 4. We use real world data of New York and Germany consisting of a railway and a foot network. Finally, we summarize the results of this work in Chapter 5.

2. Foundations

In Section 2.1 we describe the most basic concepts and set the notation. In Section 2.2 we describe the graph models. In Section 2.3 we then look at Dijkstra's Algorithm and explain profile queries in Section 2.4. As example for algorithms that compute profile queries in unimodal networks we describe a label-correcting algorithm [Dea99] and a label-setting algorithm [DKP10]. Finally, we describe how to model restrictions when switching networks by solving the Label Constraint Shortest Path Problem in section 2.5.

2.1 Preliminaries

We recap some basic concepts that will be used throughout this work: graphs, regular languages, and finite automata

2.1.1 Graphs

A *directed graph* is a tuple $G = (V, E)$ where V is a finite set of *nodes* and $E \subset V \times V$ a finite set of *edges*. An edge e from node v to w is denoted by $e = (v, w)$. Two nodes v, w are called *neighbors* when there is an edge $(v, w) \in E$ or $(w, v) \in E$. An edge $e = (v, w)$ is an *outgoing* edge of v. The node v is then the *tail* and w the *head* of e. For a node $v \in V$, $\Gamma(v)$ denotes the set of all neighbors of v. The *backward graph* $G^* = (V, E^*)$ is obtained by flipping all edges, i. e. for every $(v, w) \in E$ we insert an edge (w, v) into E^*. A *node coloring* is a function $c : V \to \mathbb{N}_0$, where the elements in $\mathbb{N}_0$ are interpreted as colors. A *valid node coloring* is then a node coloring where no two neighbored nodes have the same color, more formally:

$$\forall v \in V : \forall w \in \Gamma(v) : c(v) \neq c(w).$$

An *edge weight* is a function f assigned to an edge. We write $f(e)$ for the function of edge e, $f(e)(\tau)$ is the value of $f(e)$ when evaluated at τ. When evaluating a constant function, the notation $f(e)$ is used both for the function of e and its constant value. For two functions f, g two operations are defined, the *link operation* $f \oplus g$, denoted $f + g$ if f and g are constant, and the *merge operation* $\min(f, g)$ (For more details, see Chapter 2.2.1). Edges with constant functions are called *time-independent edges*, edges with non-constant functions *time-dependent edges*. A graph consisting only of time-independent edges is a *time-independent graph*, otherwise it is a *time-dependent graph*.

A *path* P in G is a sequence of nodes $[v_1, v_2, ..., v_k]$ so that for each $1 \leq i < k$ holds: $(v_i, v_{i+1}) \in E$.

The *length function* $len(P)$ of a path $P = (v_1, ..., v_k)$ is defined as

$$len(P) := \sum_{i=1}^{k-1} f(v_i, v_{i+1}) = f(v_1, v_2) \oplus f(v_2, v_3) \oplus \ldots \oplus f(v_{k-1}, v_k).$$

For nodes $v, w \in V$, the *shortest path function* $dist(v, w)$ is defined as

$$dist(v, w) := \min(len(P_1), len(P_2), \ldots, len(P_k))$$

where $P_1, P_2, \ldots P_k$ are all paths starting with v and ending with w.

2.1.2 Pareto-Optimality

Consider a set $\mathcal{S}$ of n-tuples. For each $x = (x_1, x_2, ..., x_n) \in \mathcal{S}$ and $y = (y_1, y_2, ..., y_n) \in \mathcal{S}$ we can compare x_i and y_i for $1 \leq i \leq n$. A tuple $x = (x_1, x_2, ..., x_n) \in \mathcal{S}$ dominates a tuple $y = (y_1, y_2, ..., y_n) \in \mathcal{S}$ when for all i with $1 \leq i \leq n$, x_i is better than or equal to y_i, and for at least one j x_j is better than y_j. Tuples that are not dominated by any other tuple in $\mathcal{S}$ are Pareto-optimal. A set consisting only of Pareto-optimal solutions is called *Pareto-set*. In this work we especially consider tuples of departure times τ_{dep} and arrival times τ_{arr} of journeys. A journey A then dominates a different journey B if the departure time of A is later or at the same time than that of B and the arrival time of A is earlier or at the same time than that of B.

2.1.3 Languages and Automata

Languages. For a finite set of symbols Σ, called an *alphabet*, a *word* with length k is defined as a sequence $[\sigma_1, \sigma_2, ..., \sigma_k]$ with $\sigma_1, ..., \sigma_k \in \Sigma$, shortly denoted as $\sigma_1\sigma_2...\sigma_k$. There is a special word, the empty word ϵ that has length zero. The concatenation of two words $w_1 = \sigma_0...\sigma_k$ $w_2 = \sigma_{k+1}...\sigma_l$ is defined as $w_1 \cdot w_2 = \sigma_0...\sigma_k\sigma_{k+1}...\sigma_l$, that is, we append w_2 on w_1. A word w concatenated with ϵ or ϵ concatenated with a word w results in the word w.

A not necessarily finite set L of words over Σ is called a language over Σ. Its i'th power is recursively defined:

$$\Delta(\tau_1, \tau_2) = \begin{cases} L^0 := \{\epsilon\}, & \text{if } i = 0 \\ L^i := \{wv \mid w \in L^{i-1} \text{ and } v \in L\}, & \text{otherwise} \end{cases}$$

The Kleene-Closure of a language L is then defined as

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

For two languages $L_1, L_2 \subseteq \Sigma^*$ the concatenated language $L_1 \cdot L_2$ is obtained by $L_1 \cdot L_2 := \{vw \mid v \in L_1 \text{ and } w \in L_2\}$. Note that this operation is not commutative. A special subset of languages are the *regular languages*. A regular language is recursively defined: The language containing no words is regular. For each $\sigma \in \Sigma$ the 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.

Finite Automata. A non-deterministic finite automaton $\mathcal{A}$ is a 5-tuple $\mathcal{A} := (Q, \Sigma, \delta, S, F)$, where Q is a finite set of states, Σ is an alphabet, $S \subseteq Q$ is a set of initial states, $F \subseteq Q$ is a set of final states, and δ is the transition function $Q \times \{\Sigma \cup \epsilon\} \to P(Q)$.

This automaton can also be depicted by its transition graph. The nodes of this graphs are all states in Q. Initial states are marked by an incoming edge-tip and final states are twin-framed. For two states q_i and q_j and a symbol $\sigma \in \Sigma$ an edge (q_i, q_j) labeled by σ

is added if and only if $q_j \in \delta(q_i, \sigma)$. For $\delta(q_i, \epsilon) \neq \emptyset$ an edge labeled ϵ from q_i to every state $q_j \in \delta(q_i, \epsilon)$ is added. These edges are called *ϵ-transitions*. An automaton without any ϵ-transition is called *ϵ-free*.

For a language $L \subseteq \Sigma^*$ a word $w \in L$ is *accepted by* $\mathcal{A}$ if there is a path in the transition graph starting at an initial state $q_0 \in S$ and leading to a final state $q_f \in F$, whereby the subsequent edges on the path are labeled by the subsequent symbols of w. Along the way, ϵ-transitions may be taken at any time. If no such path exists, the word is *rejected*. A language L is accepted by an automaton $\mathcal{A}$ if every $w \in L$ is accepted by $\mathcal{A}$.

Two automata $\mathcal{A}$ and $\mathcal{B}$ are called *equivalent* if $\mathcal{A}$ and $\mathcal{B}$ accept the exact same words.

For every regular language L there exists a non-deterministic finite automaton $\mathcal{A}$ that accepts a word w if and only if $w \in L$. On the other hand, all words accepted by an arbitrary automaton form a regular language [Kle56]. Therefore, regular languages can be described by finite automata.

2.2 Models

In this Section, we first show how we use piecewise linear functions, then present the different networks, and finally we show how they are merged into a multimodal network.

2.2.1 Piecewise Linear Functions

We use edges to model segments like streets or railway connections between two different locations (see Chapter 2.2.2 and 2.2.3 for more details). There are two types of segments we have to distinguish. Those which can be used at any time (like a street), and those which can be used only according to a schedule (like a railway connection). For the latter, both the waiting time for the next transport and the actual traveltime have to be considered [PSWZ08].

In the context of route planning we can use a *time-dependent* approach to model the traveltime in timetable networks. The idea is to have a function $f\colon \mathbb{R}_0^+ \to \mathbb{R}_0^+$ that maps the departure time to the traveltime. We only consider periodic schedules with a period time Π. Each journey is exactly the same one period later. We therefore only consider periodic functions, so for each $\tau \in \mathbb{R}_0^+$ following equation must hold:

$$f(\tau) = f(\tau \bmod \Pi).$$

We only want to find Pareto-optimal paths. For our functions, this means that the *FIFO-property* holds:

$$f(\tau_1) + \tau_1 < f(\tau_2) + \tau_2, \tau_1, \tau_2 \in \mathbb{R}_0^+, \tau_1 < \tau_2.$$

The lower bound $\underline{f}$ of a function f is defined as $\underline{f} := \min_{\tau \in \mathbb{R}_0^+} f(\tau)$.

The upper bound $\overline{f}$ of a function f is defined as $\overline{f} := \max_{\tau \in \mathbb{R}_0^+} f(\tau)$.

A periodic function $f\colon \mathbb{R}_0^+ \to \mathbb{R}_0^+$ is called *piecewise linear* if it consists of a finite number of segments of linear functions. In networks restricted by a schedule, a time-dependent network, the traveltime across a segment consists of the actual time it takes to get to the end of the segment and the waiting time. The function consists of a finite set of *connection points* B. Each connection point $p \in B$ consists of its departure time and its function value, denoted $p := (\tau, f(\tau))$. We can then compute $f(\tau)$ of an arbitrary time τ by following formula:

$$\Delta(\tau_1, \tau_2) = \begin{cases} f(\tau) = \tau_i - \tau + f(\tau_i), & \text{if next } \tau_i \geq \tau \text{ exists} \\ f(\tau) = \Pi - \tau + \tau_i + f(\tau_i), & \text{otherwise} \end{cases}$$

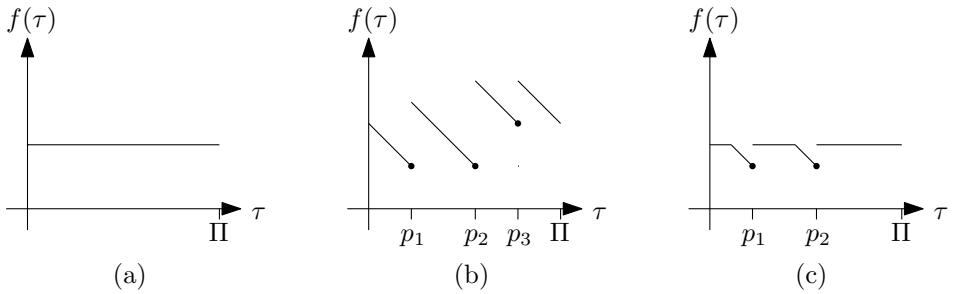


Figure 2.1: A constant function (a) and a non-constant function with three connection points (b) result in a mixed function when merged (c)

A function we can evaluate in this way is called a *non-constant function*. In time-independent networks (like road networks), we only use constant functions. Evaluation is easy then, since we just return the constant value.

There are two important operations to be performed on piecewise linear functions [Dea99].

Link. The *link operation* between two functions f,g is defined as follows:

$$f \oplus g = f + g \circ (f + id)$$

with *id* being the identity. If both functions are constant, this simplifies to $f + g$. The formula means that we evaluate each $\tau \in \mathbb{R}_0^+$ with f to get arrival times $f(\tau)$ and then see which traveltime must be added when we evaluate each of this arrival times with g. Thus, the link operation is useful in computing traveltimes over multiple segments. Note that this operation is neither commutative nor associative.

Merge. The *merge* or *minimum operation* between two functions f,g is defined as:

$$\min(f, g)(\tau) = \min(f(\tau), g(\tau)) \text{ for each } \tau \in \mathbb{R}_0^+.$$

When computing the merge operation between a constant function c and a non-constant function h, neither the evaluation rule of the constant nor that of the non-constant functions can be applied directly to the result function f. Such a function is called a *mixed function*, see Figure 2.1. A mixed function f is described by both c and h, and $f(\tau) = \min(c(\tau), h(\tau))$ for $\tau \in \mathbb{R}_0^+$. Note that if $c \geq \overline{h}$, f can be described only with h, and if $\underline{h} \geq c$, f can be described only with c.

For two functions f, g, $f < g$ holds if and only if $\min(f, g) = f$.

2.2.2 Road and Foot Network

Our road and foot networks are modeled as directed graphs. The junctions are modeled as nodes. Between two nodes v, w there is an edge (v, w) if and only if the junction associated with w can be reached via the junction associated with v over exactly one segment. Note that this means that if a segment can be used in both directions, two edges are inserted. This is needed to adequately model one way streets and streets with separate lanes in both directions. The weight of an edge is a constant function, as in [DPW09, DPW12b]. The constant assigned to this function is then the average traveltime on the associated segment, computed by the average traffic speed on this type of segment times its geographical length. For a foot network, the average traffic speed is the average speed of a pedestrian.

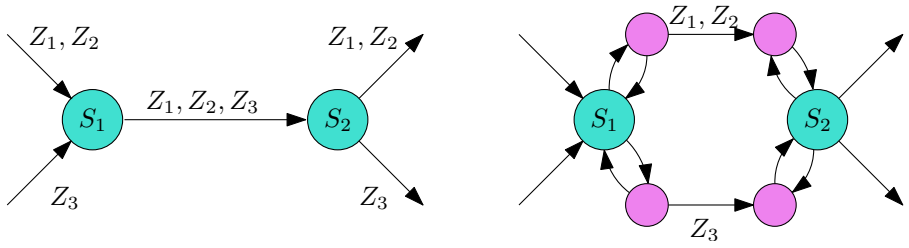


Figure 2.2: The simple model (left) and the same timetable in the realistic model (right). The stations are denoted by S_1, S_2, the trains by Z_1, Z_2, Z_3 where they take an edge [PSWZ08].

2.2.3 Rail Network

The input data of a rail networks is a *timetable*, which contains the complete schedule of all trains in the network. More formally, a traffic timetable is a tuple (C, B, Z, Π), where B is a set of *stations*, Z is a *set of trains*, Π is the *periodicity* of the timetable and C is a *set of elementary connections*. An *elementary connection* c is a tuple $c := (z, S_1, S_2, \tau_1, \tau_2)$ where $z \in Z$ is the train taken by using this connection, $S_1 \in S$ is the station from where the train z starts, $S_2 \in S$ the station where the train z arrives with no stop in between from S_1, $\tau_1 < \Pi$ the departure time from S_1 and $\tau_2 < \Pi$ the arrival time at S_2. The functions $z(c), S_1(c), S_2(c), \tau_1(c), \tau_2(c)$ yields the respective parameter of c. Moreover, the function $transfer(S)$ returns the transfer time of station S. The traveltime of a connection $\Delta(\tau_1, \tau_2)$ can be computed by:

$$\Delta(\tau_1, \tau_2) = \begin{cases} \tau_2 - \tau_1, & \text{if } \tau_2 > \tau_1 \\ \Pi - \tau_1 + \tau_2, & \text{otherwise} \end{cases}$$

In other words, the traveltime is normally computed when the connection starts and ends in the same period, otherwise the periodicity Π has to be considered. This formula only applies if there is no elementary connection with a longer traveltime than the periodicity. In our instances, there are no such elementary connections.

From this timetable, we generate a graph. Moreover, we have to model that transfers between trains take time. There are different methods to do this as discussed in [PSWZ08], with the time-dependent approach being the most feasible. This approach can also be used to model realistic transfers.

When disregarding transfer times the modeling is similar to the construction of the road network and results in the *simple model*. Each station is modeled as a node. If there is at least one elementary connection $c := (z, S_1, S_2, \tau_1, \tau_2)$ connecting the nodes v, w of S_1 and S_2, an edge (v, w) is inserted. The edge weight is a piecewise linear function (see Chapter 2.2.1), where we insert one connection point $p = (\tau_1, \Delta(\tau_1, \tau_2))$ for each such connection c.

For a model with realistic transfer times, we do not model these stations with exactly one node. First, for each station $S \in B$ we create a *station node*. At these nodes we can change between *train routes*. The set of train routes is denoted by R. Each train route $r \in R$ is a maximal subset of Z containing only trains following the exact same sequence of stations $[S_1, ..., S_k]$. For each of these train routes r that follow the sequence of stations $[S_1, ..., S_k]$ a *route node* r_i is inserted for every station S_i. Edges are inserted along this route, more formally for $0 < i < k$ an edge $e = (r_i, r_{i+1})$ is added. As in the simple model, the edge weight is a piecewise linear function. For each elementary connection c with $c(z) \in r, S_1(c) = r_i, S_2(c) = r_{i+1}$ a connection point $p(\tau_1(c), \Delta(\tau_1(c), \tau_2(c)))$ is added to the function. Each route node r is then connected with its corresponding station node

S in the following way: We insert an edge (r, S) to which we assign the constant weight 0. This way, we do not have a transfer time when we only arrive at S. We also insert an edge (S, r) that has a constant weight $transfer(S)$. This way, changing a train route takes exactly $transfer(S)$ time. For a visualization, see Figure 2.2.

This approach leads to a lot of route nodes, typically between 5 and 16 per station. To reduce this number, the *color model* was introduced in [DKP10]. The main idea is that the strict separation between the train routes is not really needed in every case. If a train z_1 arrives at a Station S and the next train z_2 is not missed due to transfer times, the transfer does not need to be modeled because it makes no difference whether time is lost due to walking or waiting. The same applies when z_2 is missed even without transfer times. Only when neither the former nor the latter applies for any of the connections containing z_1 and z_2 they need to be modeled by different route nodes. If tested for all trains running through a node s, this induces an undirected conflict graph $G_{conf}(S) = (V_{conf}(S), E_{conf}(S))$. The node set $V_{conf}(S) \subseteq Z$ consists of all trains running through S, i. e. for a train $z \in Z$ there exists an elementary connection $c \in C$ with $Z(c) = z$, $S_1(c) = S$ or $S_2(c) = S$. An undirected edge $\{Z_i, Z_j\}$ is inserted if and only if Z_i and Z_j are conflicting. A valid node coloring (see Chapter 2.1.1) on $G_{conf}(S)$ then induces a set of route nodes where no conflicting trains are modeled on the same route node by simply modeling only those trains that have the same color in $G_{conf}(S)$ on the same route node. To get a minimal number of route nodes per station, a minimal valid node coloring has to be computed. Although this is NP-hard, an approximation algorithm is good enough to reduce the amount of route nodes by a factor 6 to 12 on some instances [DKP10].

Note that when merging routes, the FIFO-property may not hold anymore. In this case, the train route has to be split into smaller routes that guarantee that the FIFO-property holds. Fortunately, this is rarely needed in real rail networks.

2.2.4 The Multimodal Network

The subnetworks have to be linked to a complete *multimodal network*. First, to identify them later, we assign labels of an alphabet Σ to the subnetworks. More specifically, we assign the same label to all nodes and edges of the same subnetwork. We then create *link edges*. For each station node of the rail network we search the nearest node in the road network and add a link edge between them in both directions. We assign a special label to these link edges. We use these labels in Section 2.5.

2.3 Dijkstra's Algorithm

Dijkstra's Algorithm finds a shortest path from a source node $s \in V$ to a target node $t \in V$ in a graph that has only constant, non-negative edge weights [Dij59]. The length of this shortest path is denoted as $dist(s, t)$ (see Chapter 2.1.1). Pseudocode is shown in Algorithm 1. For each node v a *label* $dist_s(v)$ denotes the tentative distance of the shortest known path from s to v. Its predecessor on one of the known shortest path to the node is denoted by $pre(v)$. In the beginning all distances are infinite and the nodes do not have a predecessor. More formally: $dist_s(v) = \infty$ and $pre(v) = null$ for each $v \in V$. Only the source node has a distance 0 to itself, $dist_s(s) = 0$. Every node can also be marked, but is unmarked in the beginning. All marked nodes are called *settled nodes*. The set of all nodes settled by the algorithm is called the *search space*. The invariant of this algorithm is that for all settled nodes no shorter path can be found. This is called the *label-setting property*. This obviously holds in the beginning when there are no settled nodes. Iteratively the unsettled node v with the smallest distance is chosen with help of a *priority queue* and all of its outgoing edges are *relaxed*. That is, for all outgoing edges (v, w) it is determined

Algorithm 1: DIJKSTRA'S ALGORITHM

Data: A unimodal graph $G = (V, E)$, edge weights f(e).
Input: Source node $s \in V$, target node $t \in V$.
Output: OUT, traveltimes and Path from s to t.

```
   // Initialization
 1 forall v ∈ V do
 2     dist_s(v) ← ∞
 3     pre(v) ← null
 4 dist_s(s) ← 0
 5 PQ.INSERT(s, 0)
   // Main loop
 6 while PQ is not empty do
 7     (v) ← PQ.DELETEMIN()
 8     MARK(v)
 9     if STOPPINGCRITERIONHOLDS() then
10         break
11     forall outgoing edges e = (v, w) do
12         dist ← dist_s(v) + f(e)
13         if dist < dist_s(w) then
14             dist_s(w) ← dist
15             pre(w) ← v
16             if PQ.CONTAINS(w) then
17                 PQ.DECREASEKEY(w, dist_s(w))
18             else
19                 PQ.INSERT((w), dist_s(w))
20 OUT ← dist_s(t)), backtrace predecessors of t, give out in reverse order
```

whether there is a shorter path to w via v. If this is the case, $dist_s(w)$ is updated and w is put in the priority queue and the predecessor of w will be set to v (line 10-18). Now, since every node with a smaller distance has already been settled and its outgoing edges have been relaxed (and there are only positive edge weights), there cannot be any better path to v, so v can be marked. The algorithm stops when the priority queue is empty, which means there are no nodes left that can be settled. Then, for every $v \in V$ it holds that $dist_s(v) = dist(s, v)$. We have computed a *one-to-all query*. Alternatively the algorithm stops when a *stopping-criterion* holds. We may stop the search when the target node has been settled. Then $dist_s(t) = dist(s, t)$ holds, we have computed a *point-to-point query*.

There are many extensions of Dijkstra's Algorithm. If there are multiple source and target nodes, and we want to find the shortest path between any source and target node, the algorithm is easily adapted by setting the label of all source nodes to zero and adding them to the priority queue. We then may stop when the priority queue is empty or the first target has been settled.

A Pareto-generalization of Dijkstra's Algorithm is to not only compute the result for one optimization criterion, but a set of Pareto-optimal solutions for multiple criterions. In this case, the label is not a single value, but a r-dimensional vector, where r is the number of optimization criterions. We now store a Pareto-set at the nodes instead of just a single label. For a node v we denote this set S_v. When relaxing an edge (v, w), we take labels from w, compute new ones of them and the edge weight, and *try to insert* the new labels at

w. We call this to *propagate* the labels over an edge. Trying to insert a label means hereby that we determine which labels of S_w are dominated. When one of the new labels is not dominated by any label of S_w, w is updated. There are two approaches to propagate labels. In the label-correcting approach, a whole Pareto-set S_v is propagated at once. Note that each node may then be settled multiple times, so the label-setting property does not hold. In the label-setting approach, we propagate only one label at a time. The label-setting property then holds, but it may be more costly to determine which labels are dominated than if we checked this for many labels at once.

2.4 Profile Queries

In a graph with only constant edge weights we search the shortest path between two nodes s and t, independently from the departure time. In time-dependent networks, the shortest path found may differ significantly for different departure times. A *time query* is then a classical s-t search for a departure time τ. Dijkstra's Algorithm can be simply adapted for such a time query: Instead of just taking the constant edge weight $f(e)$ in line 12, we have to take the value of the time when we arrive at this edge, $f(e)(\tau + dist_s(v))$.

Just finding the shortest path may not be the users only priority, though. In the time-independent case the user can deliberately choose the departure time, so the shortest traveltime is the only optimization criterion. In the time-dependent case, the shortest traveltime can be different for different departure times. Those departure times and their corresponding arrival times may be criterions for an optimal path as well, considering appointments and personal preferences of the user. However, defining mathematically just how important those criterions are compared to the shortest traveltime is difficult, even for the user himself. Because of this, we want to compute a function that maps all departure times to their shortest traveltime, so the user can choose the journey that is optimal for him. To find such a function is called a *profile query*, the resulting function is called *profile function*. We can compute such a profile query by first finding a Pareto-set of optimal journeys, i. e. all journeys that are not dominated by other journeys (see Section 2.1.2), and then using the departure times and traveltimes of these journeys as connection points of a piecewise linear function (see Section 2.2.1).

In [Dea99], a label-correcting approach for unimodal networks is presented. Instead of scalar values, whole functions are stored at the nodes, denoted $f(v)$ for a node v. Each of these functions depicts the shortest tentative traveltime to the node for all departure times. As key for the priority queue the lower bounds of the functions are used. When relaxing an edge $e = (v, w)$, instead of adding scalar values, they link functions: $f(v) \oplus f(e)$. If the resulting function is better than $f(w)$ at only one time point, w is updated in the priority queue. When finished, the function at the target node depicts the shortest traveltime for every departure time. This algorithm is the basis for the Function Algorithm presented in Chapter 3.

In [DKP10], Delling et. al. take a label-setting approach for rail networks. They compute the shortest traveltimes only for departure times of transports one can take from the source station. For each node the tentative arrival time for each of these departures is stored. The tentative arrival times are then successively settled as in Dijkstra's algorithm. When one can skip a transport at the source station and still arrive earlier than when he would have taken it that transport is ignored. The overall traveltime for an arbitrary departure can then be computed by adding the waiting time for the next transport to the shortest traveltime at its departure time. Unfortunately, in a multimodal network there can also be road graphs, where we can start at any time so this approach is not feasible for us.

Note that in the multimodal case, we have both time-dependent and time-independent subnetworks. Shortest paths that are solely in time-independent networks do not have a departure time. We have to consider this when developing our algorithms in Chapter 3.

2.5 Label Constraint Shortest Path

Simply performing a profile query on the model introduced in Section 2.2 is not sufficient for the multimodal case. This is because paths which are undesirable or unrealizable in practice are computed. It may happen that a small amount of time is saved by extensively jumping between rail and road networks. This is not only annoying for the user, but may be impossible since he has to get a new car every time he switches to the road network. Therefore the possibility to change between networks unlimitedly has to be restricted. In [DPW09], the authors developed a solution for time queries that can easily be extended for profile queries. First a unique label for each network is assigned to each node and edge within this network (see 2.2.4). Then the Label Constraint Shortest Path Problem is solved on the resulting graph, using regular languages and finite automata (see 2.1.3).

Label Constraint Shortest Path Problem. Given an alphabet Σ, a language $L \in \Sigma^*$, a directed graph $G = (V, E)$ with edge weights and Σ-labeled edges, we ask for the shortest Path P from s to t whereby the sequence of labels of the edges form a word of L. More formally, for a Path $P = [v_0, ..., v_k]$ it must hold:

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

Regular languages are sufficient to adequately model the restrictions in our multimodal network [DPW09]. In [BJM00], Chris Barrett et al. established that the Label Constraint Shortest Path Problem is solvable in polynomial time for regular languages and present an algorithm for doing so. The first step is constructing a *product network* that combines the graph and a finite automaton $\mathcal{A}$ that describes the desired language L.

Product Network. Given a Σ-labeled graph $G = (V, E)$ and a non-deterministic automaton $\mathcal{A} := (Q, \Sigma, \delta, S, F)$, the product network is defined as $G^\times = (V^\times, E^\times)$, where $V^\times$ consists of product-nodes (v, q) with $v \in V$ and $q \in Q$. A product-edge $e^\times = ((v_1, q_1), (v_2, q_2))$ is in $E^\times$ if and only if $e = (v_1, v_2) \in E$ and for $\sigma = label(e)$ holds: $q_2 \in \delta(q_1, \sigma)$. The edge weight of $e^\times$ is the edge weight of e and label $(e^\times)$ is set to σ. The resulting graph is unimodal.

The Label Constraint Shortest Path Problem can then be solved by an adaption of Dijkstra's Algorithm. First, the (non-deterministic) finite automaton $\mathcal{A} := (Q, \Sigma, \delta, S, F)$ describing L is constructed. With this automaton, the product network $G^\times$ is created. Finally, a query on $G^\times$ with a multi-source, multi-target version of Dijkstra's Algorithm is performed, where source and target node sets are:

$$S := \bigcup_{q_s \in S} (s, q_s) \text{ and } T := \bigcup_{q_f \in F} (s, q_f)$$

The problem with this algorithm is that the product network consumes a lot of space, to be specific $O(|G| \cdot |A|)$. in [BBH+08] it is proposed to store both structures separately. We store information for each node/state combination of the original graph. When we relax an edge $e = (v, w)$ using an information stored in state q, we have to check whether the information at any node/state combination (w, q') has to be updated where $q' \in \delta(q, label(e))$. This way, we compute the edges of the product network implicitly and save memory space. In the following, we call this adaption of Dijkstra's Algorithm *LC-Dijkstra*.

Both algorithms for profile queries presented later in this work use the Label Constraint Shortest Path Problem in a very similar way to model restrictions for the sequence of modes of transports.

3. Algorithms

In this chapter we present our two algorithms that compute profile queries on multimodal networks. Both use a regular language to model restrictions for the sequence of modes of transports, and both have to deal with the fact that we have both time-dependent and time-independent subnetworks. First, we present the Function Algorithm, a label-correcting algorithm. It is based on an augmentation of Dijkstra's Algorithm, where we propagate complete functions through the network in place of scalar values [Dea99]. We show that the loss of the label-setting property in this algorithm can lead to drawbacks. We then present the Label Algorithm, a label-setting algorithm in which the label-setting property holds.

3.1 Function Algorithm

The *Function Algorithm* combines a label-correcting approach to compute profile queries ([Dea99], see Chapter 2.4) and an approach to compute multimodal time queries ([BJM00], see Chapter 2.5). The former uses a version of Dijkstra's Algorithm that propagates whole functions over the network instead of scalar values. The latter implicitly computes a product network of a graph with Σ labeled edges and a finite automaton $\mathcal{A} = (Q, \Sigma, \delta, S, F)$. This automaton represents a regular language L that describes the valid sequences of taken modes of transports. The algorithm then solves the Label Constraint Shortest Path-Problem (see Chapter 2.5).

Pseudocode for the Function Algorithm is shown in Algorithm 2. Input to our algorithm is the graph of the multimodal network, the finite automaton, as well as the source node s and the target node t. First, we augment Dijkstra's Algorithm to propagate whole functions. We do not store scalar distances at the nodes. Instead, we store a piecewise linear function for each node/state tuple $(v, q) \in V \times Q$. Note that this function may be mixed, as we have both time-dependent and time-independent graphs. It contains the tentative shortest traveltimes from s to v whereby we are in state q when using the labels of the taken edges as input for the automaton. For a tuple $(v, q) \in V \times Q$ this function is denoted $f(v, q)$. As key for such a tuple we use the lower bound of its function, denoted $\underline{f}(v, q)$. This choice of key can lead to issues as discussed later. But there are few other choices of what we can use to map a function to a scalar value, and this choice of key at least allows for a reasonable stopping criterion as shown below. The label of an edge is denoted by $label(e)$, the edge weight of an edge is also a piecewise linear function, denoted by $f(e)$. Instead of

Algorithm 2: FUNCTION ALGORITHM

Data: A multimodal graph $G = (V, E)$, edge weights, a finite automaton $\mathcal{A} = (Q, \Sigma, \delta, S, F)$ representing a regular language $L \subseteq \Sigma^*$.
Input: Source node $s \in V$, target node $t \in V$.
Output: $f(out)$, profile function from s to t.

```
// Initialization
forall v ∈ V, q ∈ Q do
    f(v, q) ← ∞
forall q_s ∈ S do
    f(s, q) ← 0
    PQ.INSERT((s, q), 0)

// Main loop
while PQ is not empty do
    (v, q) ← PQ.DELETEMIN()
    if STOPPINGCRITERIONHOLDS() then
        break
    forall outgoing edges e = (v, w) do
        forall states q' ∈ δ(q, label(e)) do
            f_new ← f(v, q) ⊕ f(e)
            if NOT f(w, q') ≤ f_new then
                f(w, q') ← min(f(w, q'), f_new)
                if PQ.CONTAINS((w, q')) then
                    PQ.DECREASEKEY((w, q'), f̲(w, q'))
                else
                    PQ.INSERT((w, q'), f̲(w, q'))
forall states q ∈ F do
    f(out) ← min(f(t, q), f(out))
```

summing and comparing scalar values, we use the corresponding operations of piecewise linear functions (see Chapter 2.2.1).

We initialize the algorithm by adding the source node in combination with all initial states of the finite automaton to a priority queue PQ. In the main loop, we get the node/state tuple (v, q) with the smallest key. We then iterate over every edge of the product network. These transitions are implicitly computed in Line 10 and 11. Here, we see which states we reach in the automaton from state q for $label(e)$. We then link $f(v, q)$ and $f(e)$, resulting in a function $f_{distvia}$ that depicts all tentative shortest travel times over e. If for even one time point τ it holds that $f_{distvia}(\tau) < f(w, q')(\tau)$ we have to update (w, q'). An update does not replace the function of (w, q'), instead, $f_{distvia}$ and $f(w, q')$ are merged. By doing so, all not Pareto-optimal connection points of $f(w, q')$ are deleted. We stop when a stopping criterion holds or the priority queue is empty. The profile function from s to t where only valid sequences of taken transports are considered can then be computed by merging all functions $f(t, q)$, where q is a final state of the automaton.

Stopping Criterion. We may stop as soon as the profile function $f(out)$ can not change anymore. This is the case when $PQ.MIN \geq \overline{f}(out)$ [DW09]. As $f(out)$ is computed by merging all functions $f(t, q)$ with $q \in Q$, our stopping criterion is:

$$PQ.MIN \geq \min_{q \in F}(\overline{f}(t, q))$$

Drawbacks. The label-setting property does not hold as nodes may be settled multiple times. The key of the node/state tuple we extract does not even always increase after every iteration of the main loop. For an example, consider two functions f and g of two node/state combinations (v, q) and (w, q'). The connection points of f are $b_1 = (0, 0)$ and $b_2 = (1000, 500)$, the ones of g are $c_1 = (0, 20)$ and $c_2 = (1000, 40)$. The lower bounds of f and g are then 0 and 20 respectively. We extract (v, q) before (w, q'). After that, it is still possible to improve b_2, and we would then reinsert (v, q) into the priority queue. The lower bound of f is still 0, however. The key is not that meaningful in this case, since the key did not change although b_2 was improved.

We present a scenario in which the algorithm is not working well due to the explained shortcomings. For a visualization, see Figure 3.1. We start at a station in the railway network. Only two trains T_{day} and T_{night} depart at this station in two different directions. The train T_{day} departs at noon, T_{night} at midnight. They both take the same time to reach the foot network, T_{day} arrives at node a, T_{night} at node b. For simplicity, we assume that the foot network is a regular grid where all edges have the same small constant edge weights. The nodes a and b are far apart, but because T_{day} and T_{night} depart at very different times, it is still fastest to travel to b by taking T_{day} and then walking when departing at noon, and to travel to a by taking T_{night} and then walking when departing at midnight. This still allows for large distances between a and b. The two connection points belonging to the two trains are propagated circular in manhattan distance around their arrival points, leading to two groups of connection points. Until the groups touch for the first time, no node is settled multiple times. In a label-setting algorithm, the two groups would now begin to overlap. Instead, the lower bound of the function is used as key, but the lower bound does not change for any function in this scenario. Thus, the already settled nodes will be settled again a second time - and all at once. We propagate functions consisting of not one, but two connection points, even if one of them does not change anything when propagated. Only after all these nodes are settled the algorithm continues to settle new nodes. Note that this effect depends on how far apart the arrival points of the trains are. Also, worse scenarios can be imagined where there are more than two trains with sufficiently different departure times.

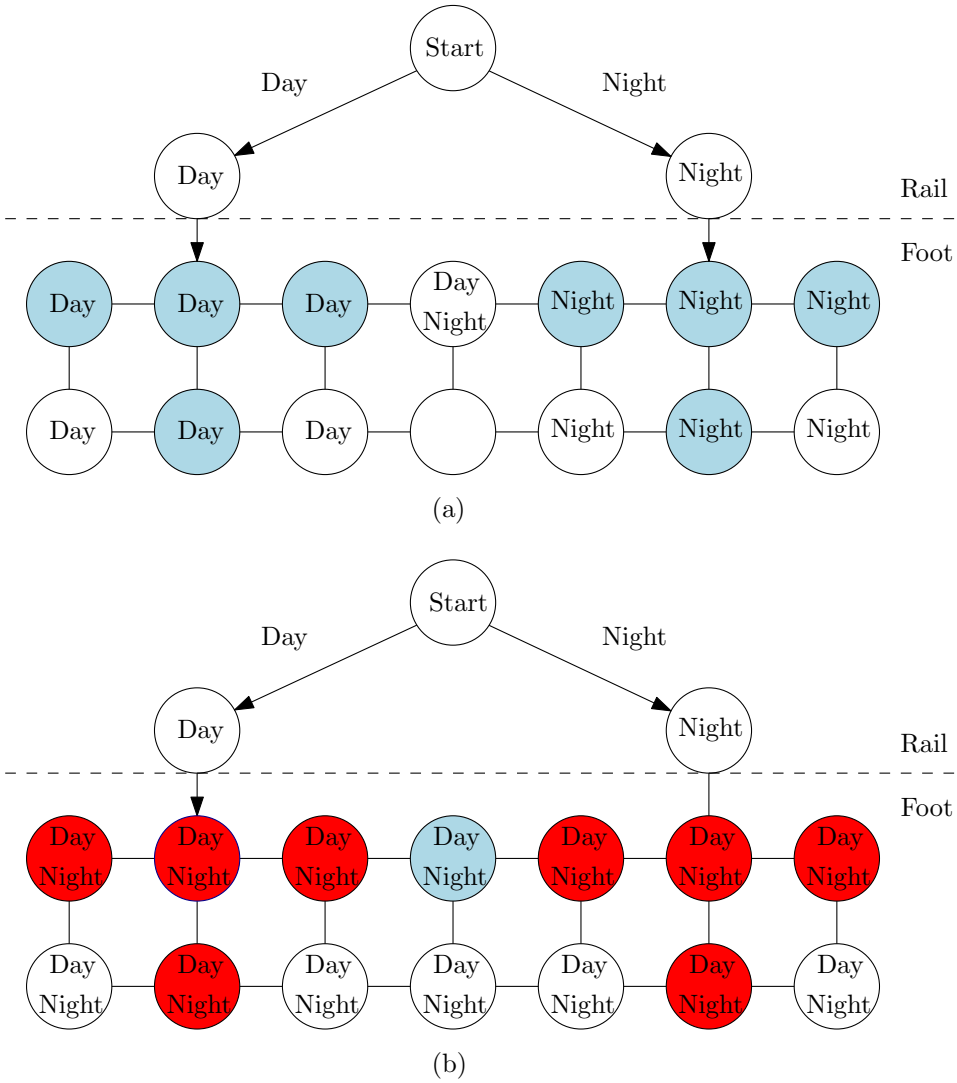


Figure 3.1: Two trains start at noon and midnight, which leads to the propagation of two groups of connection points, denoted Day and Night. They travel the same time and reach the foot network that is assumed to be a uniform grid, all edges have the same small constant edge weights. From there the groups are propagated until they touch for the first time, leading to the state depicted in Subfigure (a). The bright blue nodes are all foot nodes that are settled once. The two groups start at very different times, so they do not dominate each other. The nodes around the node where the groups touched are updated. As key the lower bound of a function is used, so the nodes are settled again at once with a function double the size. This process repeats itself until all settled nodes are settled a second time, which is depicted by the dark red color. This leads to the state depicted in Subfigure (b).

3.2 Label Algorithm

In this section, we present a label-setting algorithm, the *Label Algorithm*, in which we try to overcome the shortcomings of the Function Algorithm. We propagate single labels, so that the label-setting property holds. Therefore we first introduce some data structures and definitions we need in order to explain the algorithm. We then present the algorithm in detail.

3.2.1 Data Structures

A *single label* is a tuple $sl := (\tau_{dep}, \tau_{tra}, marked)$ where $\tau_{dep} < \Pi$ is the departure time of a journey, τ_{tra} its current traveltime at this point and $marked$ signaling whether the single label has been settled yet. The arrival time is $\tau_{arr} = \tau_{dep} + \tau_{tra}$. Note that the arrival time may be bigger than Π, since the arrival can be in one of the following periods. The values τ_{dep} and τ_{arr} may be undefined, denoted $\perp$, if they are not applicable. This is the case with journeys that take place solely in time-independent networks where there are no scheduled departures. Hence we call this kind of labels *pure independent labels*. The function $isPure(sl)$ returns true if and only if sl is a pure independent label, $\tau_{tra}(sl)$,$\tau_{dep}(sl)$, $\tau_{arr}(sl)$ and $marked(sl)$ return the respective parameter of the single label. Two single labels A and B are considered *equivalent* if and only if $\tau_{dep}(A) = \tau_{dep}(B)$ and $\tau_{tra}(A) = \tau_{tra}(B)$. We define *domination* between single labels. This is almost as defined in Chapter 2.1.2, but we have to consider pure independent labels. For two single labels A and B we have to differ between four cases:

1. The single labels A and B are pure independent labels. Then A dominates B if and only if $\tau_{tra}(A) < \tau_{tra}(B)$.
2. The single labels A and B are both not pure independent labels. Then we have two cases:
   a) It holds that $\tau_{dep}(A) \geq \tau_{dep}(B)$. Then A dominates B if and only if A and B are not equivalent and $\tau_{arr}(A) \leq \tau_{arr}(B)$.
   b) It holds that $\tau_{dep}(A) < \tau_{dep}(B)$. This indicates that the journey described by A begins earlier than B. However, A can still dominate B if B ends after A, but in another period. More precisely, A dominates B if $\tau_{arr}(B) > \tau_{arr}(A) + \Pi$.
3. The single label A is a pure independent label, B is not. Then A dominates B if and only if $\tau_{tra}(A) < \tau_{tra}(B)$. This is because the journey of A can start at any time, also at $\tau_{dep}(B)$. Then this case can be reduced to case 2.
4. The single label A is not a pure independent label, B is. In this case we define that A cannot dominate B. We could define that A dominates B when $\Pi + \tau_{tra}(A) > \tau_{tra}(B)$ where Π is the maximal possible waiting time, the whole period. However, this case is very rare and costs computing time.

For the key of a single label sl, denoted $key(sl)$, the possible candidates are τ_{dep}, τ_{arr} and τ_{tra}. While taking the arrival key may have its merits, as shown in [DKP10], the traveltime is the only value that is guaranteed to exist in every single label, so $key(sl) := \tau_{tra}(sl)$.

For a traveltime τ_{tra} and an arrival time τ_{arr} the function $dep(\tau_{tra}, \tau_{arr})$ computes the departure time $\tau_{dep} < \Pi$ by the following formula:

$$dep(\tau_{tra}, \tau_{arr}) = \begin{cases} \tau_{arr} - \tau_{tra}, & \text{if } \tau_{arr} \geq \tau_{tra} \\ \Pi - (\tau_{arr} - \tau_{tra}), & \text{otherwise} \end{cases}$$

A *node label* is a data structure that manages all single labels belonging to a specific node $v \in V$. For each state $q \in Q$ it has a Pareto-set of single labels, the *labelset*. When referring to a node label at node v or a labelset at node v and state q, we write $nodelabel(v)$ and $labelset(v, q)$ respectively. As we break ties arbitrarily, we do not allow two identical single labels in a labelset. This implies that there can be at most one pure independent single label for each labelset. We can convert a labelset into a piecewise linear function if needed. We assign the traveltime of the pure independent single label to a constant function g. All other single labels $sl_1, ..., sl_k$ are ordered increasingly by departure time and used as connection points, with $p_i := (\tau_{dep}(sl_i), \tau_{tra}(sl_i))$, yielding a function h. The complete

function f is then $f = \min(g, h)$. We write $f(v, q)$ for the function converted from the labelset at node v with state q.

There are some operations the node label must support.

Representant. Finds the labelset that contains the unmarked single label sl with the lowest key. It then returns its state q and sl in a tuple (sl, q). We write $representant(v)$ to get the representant at node v.

Insert. Inserts a single label for a state q into the respective labelset and returns whether it is successfully inserted. The single label may only be inserted if it is not dominated or equaled by any single label, and all single labels it dominates have to be deleted. We write $insert(v, q, sl)$ for inserting a single label into the labelset of node v with state q. For convenience, when we create a new single label and insert it at a node v with state q, we write insert$(v, q, \tau_{dep}, \tau_{tra})$ for departure time τ_{dep} and traveltime τ_{tra}.

Settle. In some of the presented implementations of *representant* and *insert*, it is not enough to mark a single label when it is settled at node v. Additional operations are then performed by $settle(v)$.

3.2.2 The Algorithm

We now present the Label Algorithm in detail. We propagate single labels, which have a simple key. As in the Function Algorithm, our input is the graph of a multimodal network, a finite automaton $\mathcal{A} = (Q, \Sigma, \delta, S, F)$ describing a language $L \in \Sigma^*$ and the source and target nodes s and t. For each initial state of the automaton a pure independent label with traveltime 0 is inserted into the node label belonging to s. The priority queue PQ stores nodes, the key of a node is the key of its representant. In the main loop we get the unmarked single label with the smallest key at a node v. As it is the single label with the smallest traveltime, it can not be dominated by any unmarked single label, and all marked single labels have already been propagated. So the label-setting property holds and we mark this single label. The node v may have another representant now, if so we have to update v in the priority queue. The old single label is then propagated over each outgoing edge $e = ((v, q), (w, q'))$ of the product network. We compute these edges implicitly as in the Function Algorithm (see Section 3.1). We have to differ between two cases then. If either our edge weight is a constant function or the single label is not pure independent we do this straightforward. We create a new single label with the same departure time as the old one and add the traveltime for taking the edge when we arrive there. The second case is when the single label is pure independent and the edge weight is a non-constant function. This means the journey described by the single label only used time-independent edges up to this point, and has no single departure time. This is the first time where one taking this journey has to wait for a transport. So we create a new single label sl for each connection point $p = (\tau, f(e)(\tau))$ of the edge weight. These connection points represent departures. To arrive in time we have to start the journey at $dep(\tau_{tra}(sl), \tau)$, so this is the new single label's departure time. We then take the transport directly at its departure time, so the overall traveltime is $\tau_{tra}(sl) + f(e)(\tau)$. Note that this propagation of multiple single labels can happen only once per edge, as there can be only one pure independent single label at each node, and this single label is settled during this operation.
In both cases, all newly created labels are then inserted into the labelset l at node w with the state q'. Each one is only inserted if there is no single label in l that dominates or is equivalent to the created one. If a single label was successfully inserted, we update the queue, i. e. we insert w if it was not inserted before, otherwise its key is decreased. As in the Function Algorithm, we may stop if either the queue is empty or a stopping criterion holds. The profile function can be computed by converting all labelsets at t with a final state of the automaton into a function and then merging them.

Algorithm 3: Label Algorithm

Data: A multimodal graph $G = (V, E)$, edge weights, a finite automaton $\mathcal{A} = (Q, \Sigma, \delta, S, F)$ representing a regular language $L \subseteq \Sigma^*$.
Input: Source node $s \in V$, target node $t \in V$.
Output: $f(out)$, Function of traveltimes from s to t.

```
// Initialization
forall v ∈ V, q ∈ Q do
    labelset(v, q).CLEAR
forall q_s ∈ Q do
    insert(s, q, ⊥, 0)
    PQ.INSERT(v, 0)

// Main loop
while PQ is not empty do
    v ← PQ.DELETEMIN()
    (sl, q) ← representant(v)
    settle(v)
    if Another representant(v) exists then
        PQ.DECREASEKEY(v, key(representant(v)))
    if STOPPINGCRITERIONHOLDS() then
        stop
    forall outgoing edges e = (v, w) do
        forall states q' ∈ δ(q, label(e)) do
            if isPure(sl) and f(e) is not constant then
                wasInserted ← false
                forall Connection points p = (τ, f(τ)) of f(e) do
                    wasInserted ← wasInserted or
                    insert(w, q', dep(τ_tra(sl), τ), τ_tra(sl) + f(e)(τ))
                if wasInserted then
                    PQ.UPDATE(w, key(representant(w)))
            else
                wasInserted ← insert(w, q', τ_dep(sl), τ_tra(sl) + f(e)(τ_arr(sl)))
                if wasInserted then
                    PQ.UPDATE(w, key(representant(w)))
forall states q ∈ F do
    f(out) ← min(f(t, q), f(out))
```

Stopping Criterion. As in the Function Algorithm, we may stop only if no time point of the profile function $f(out)$ can be improved. The profile function is computed as in the Function Algorithm, all edge weights are positive and the label-setting property holds, leading to the same Stopping Criterion as in the Function Algorithm:

$$PQ.MIN \geq \min_{q \in F}(\overline{f}(t, q))$$

Note that the upper bound can be easily computed by converting the labelset into a function, and this has to be done only for the target node.

A drawback of the Label Algorithm is that we have to perform the extract operation of the priority queue very often, as there are much more single labels in the Label Algorithm than there are functions in the Function Algorithm. Moreover, the running time depends highly on how the methods $representant(v)$ and $insert(v, q, sl)$ are implemented. We therefore show the approaches we have explored for implementing them.

Implementation of representant(v)

The *representant* method is called three times in the main loop. First when settling a single label, then when reinserting the node after settling the single label, and finally every time we update a node after relaxation. Therefore implementing this method is vital to the performance of the algorithm. The simplest method is using an unsorted labelset and performing a linear search through each labelset when searching a representant, though this is time expensive.

Tracking the representant. The first option is to track the representant whenever possible and only to perform a linear search if absolutely necessary. Keeping tracking up to date is no problem when inserting a single label in a labelset. We have to check whether the inserted single label has a smaller traveltime than the current representant. The only operation when we change a labelset without inserting is by calling the *settle* method. Here our current representant is marked and can therefore not be used as representant anymore. This is the only occurrence where a linear search through the labelset is unavoidable to find the new representant, but this is only necessary once in the main loop.

Sorting the labelset. The second option is to keep the labelset sorted non-decreasingly by traveltime. Since the label-setting property holds, all marked single labels are then at the beginning of the labelset, and the first unmarked single label is the representant. If we store the position of marked single labels, this operation takes only constant time. However, the *insert* method needs to be modified to keep the labelset sorted, which takes time.

Implementation of insert(v)

The *insert* method is another frequently called operation. Though mentioned only once in the algorithm, it is in the main loop and is called for every relaxed edge there. The naïve implementation of this method is to compare the inserted single label with every other single label to determine which ones are dominated. Another approach is again to sort the labelset. Sorting by traveltime achieves little, since traveltime is only a necessary criterion for domination and we have to check both whether the inserted single label dominates anything or is dominated. It is better to sort increasing by departure time. We observe that when the labelset is sorted by departure time, it is sorted by arrival time as well. Consider two single labels sl_1 and sl_2 with $\tau_{dep}(sl_1) \geq \tau_{dep}(sl_2)$. If $\tau_{arr}(sl_1) < \tau_{arr}(sl_2)$, sl_1 dominates sl_2, therefore sl_2 cannot be in the labelset. Note that a pure independent label has no departure time. Fortunately there is at most one of them in the labelset, so

we treat it as a special case, and check the domination relation with it at the beginning. When inserting a pure independent label that is not dominated by the pure independent label of the labelset, we still have to do a linear search. But this case is rare and if it occurs, the single independent label has a good chance of dominating many single labels of the labelset. In all other cases, we proceed as follows:

We can determine the position POS_{dep} that the inserted single label sl would have if we inserted it in the labelset so that the labelset is still sorted by departure time. More precisely, we try to insert it at the first position so that the departure time of sl is strictly bigger than its predecessors. To find this position we do not just do a linear search. We can assume that the departure times in the labelsets are more evenly distributed over the period the more single labels it contains. We therefore assume that the single labels are evenly distributed and interpolate a position that is likely near POS_{dep}. From there, we can search up or down to find the exact POS_{dep}. From this point, we can search the position where we would insert sl so that the labelset is still ordered by arrival time, POS_{arr}. We then have three cases:

1. For a single label sl, POS_{arr} is not at the beginning or the end of the labelset. As defined in Section 3.2.1, there are two rules for checking domination. The second rule, which handles journeys extending into another period cannot apply here, or either the first or last single label in the labelset would have been dominated already. According to the first rule however, only those single labels with a departure time bigger or even $\tau_{dep}(sl)$ can dominate sl, which means only single labels with a position p $\geq POS_{dep}$, or the predecessor of POS_{dep}, can dominate sl. Likewise, sl can dominate only single labels at position $q < POS_{dep}$. Analogously, only single labels at position $p < POS_{arr}$ can dominate sl, and sl can dominate only single labels at a position $q \geq POS_{arr}$ or at the predecessor of POS_{arr}.

This leads to following rules:

1. If $POS_{dep} < POS_{arr}$, sl is dominated by at least one single label.
2. If $POS_{dep} = POS_{arr}$, sl can dominate and can be dominated only by the label $POS_{dep} - 1$, or equal it.
3. If $POS_{dep} > POS_{arr}$, sl dominates all single labels with positions in $[POS_{arr}, POS_{dep})$. If the arrival time of the single label at $POS_{arr} - 1$ is the same as $\tau_{arr}(sl)$, the label at $POS_{arr} - 1$ is also deleted.

Note that after we have found POS_{dep}, we only need to check as much labels as we delete, and only one if we are dominated, plus the label at $POS_{dep} - 1$ if needed. We also have to delete only a contiguous area of memory which is more efficient on data structures like arrays than deleting several single entries. For examples, see Figure 3.2.

2. For a single label sl, POS_{arr} is at the end of the labelset. The single label sl cannot dominate any other single labels, except possibly the one at $POS_{arr} - 1$ if POS_{dep} is also at the end. If $POS_{dep} < POS_{arr}$, sl is dominated as in the former case. However, even if $POS_{arr} = POS_{dep}$, sl is possibly dominated by the first entry of the labelset. For an example, see Figure 3.5.

3. For a single label sl, POS_{arr} is at the beginning of the labelset. The single label sl cannot be dominated by any other single label. The single labels at positions in $[POS_{arr}, POS_{dep})$ are dominated. However, it is still possible for sl to dominate further single labels. We check what happens when the journeys belonging to all other single labels begin during the previous period. This means that Π is subtracted from their arrival times, or, equivalently but more efficiently, Π is added to the arrival time of sl. We can then search for the position of this new arrival time POS_{arrnew}, beginning at the

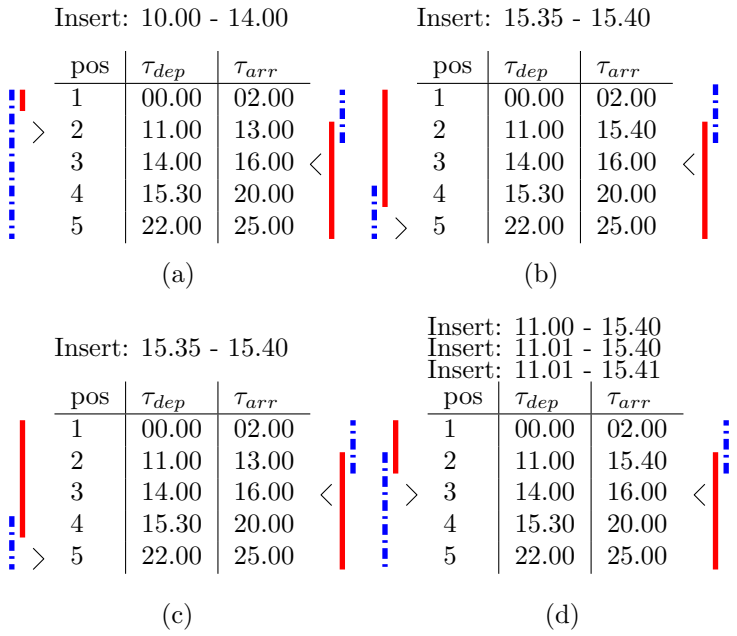


Figure 3.2: Subfigures (a)-(d) show labelsets, with the position, departure time τ_{dep} and arrival time τ_{arr}, as well single labels that should be inserted. An arrowhead marks POS_{dep} on the left hand side and POS_{arr} on the right hand side. The red line marks which single labels may be dominated, the blue dotted line which single labels may dominate the inserted one, according to POS_{dep} and POS_{arr} respectively. In Subfigure (a), the inserted label is dominated since the blue lines overlap in more than one single label. In Subfigure (b), the red line overlaps for single labels 2-4 which are dominated. Subfigure (c) shows hat the predecessor of $\min(POS_{dep}, POS_{arr})$ has to be checked as a special case. This case is almost like (b), but single label 2 is not dominated. Subfigure (d) demonstrates this even better, as the three inserted single labels produce the same POS_{dep} and $POSarr$, but the first is not inserted because it equals single label 1, the second dominates it, and the third is inserted without changing anything.

end of the labelset. The single label *sl* then dominates each single label at the positions $p \geq POS_{arrnew}$. Again, *sl* might dominate the single label at $POS_{arrnew} - 1$. We again have to check only as many single labels as we delete, plus the one at $POS_{arrnew} - 1$ and have to delete two contiguous areas of memory. Note that we never have to go back two periods in time, because then *sl* dominates every single label even after going back only one period and we can stop. For an example, see Figure 3.5.

Implementing Both Representant and Insert

Preliminary results showed that implementing *representant* by sorting the labelsets by traveltime results in a higher speedup than just tracking the representant. Unfortunately, this is not compatible with the non-naïve implementation of *insert*. We clearly achieve the highest speedup when we combine implementing *insert* by sorting the labelsets by departure time and tracking the representant, so we focus on this approach in the rest of the work.

3.3 Improvements

Both algorithms use ideas similar to those used to solve the Label Constraint Shortest Path-Problem Problem (see Chapter 2.5). In practice, there are some instances where this

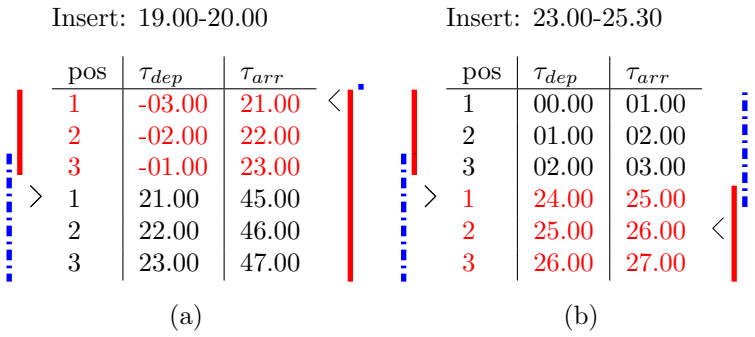


Figure 3.3: The notation is as in Figure 3.2. Single labels set in red are single labels of the labelset that are set into another period. Subfigure (a) shows how single labels are dominated when set into a former period. Subfigure (c) shows how a single label in a later period dominates the single label that should be inserted.

approach can be optimized. First, we see that the automata themselves can be optimized, then we improve the running time by using a special backward search in some cases.

3.3.1 Domination Between States

Both the Label and Function Algorithm only eliminate single labels and connection points that dominate each other in the same state. In general, this is necessary for correctness, because being in a certain state means that only certain modes of transport may be used in the future. However, this is not always the case. Consider the automaton in Figure 3.4, where a journey begins in the foot network, goes over the rail network and finally returns into the foot network, and every state is final. In the last state, one can only use foot edges. However, this would also lead to valid paths when one was in the first state. Hence, when a single label or connection point of the first state dominates one in the last state, the single label or connection point in the last state can be deleted. We can generalize this: If a part of a journey begins in state B, and all valid paths from this point on are also valid when the same part of the journey began in state A, then single labels and connection points in state A may dominate those in state B. We say, state A *dominates* state B. More formally:

Definition 1. *Given an Automaton $\mathcal{A} = \{Q, \Sigma, \delta, S, F\}$ and two states $p, q \in Q$, consider the two automata $\mathcal{B} = \{Q, \Sigma, \delta, p, F\}$ and $\mathcal{C} = \{Q, \Sigma, \delta, q, F\}$. Then p dominates q if and only if every word $w \in \Sigma^*$ that is accepted by $\mathcal{C}$ is also accepted by $\mathcal{B}$.*

This can be implemented by simply adjusting the automaton. When A dominates B, adding an ϵ-transition from A to B results in an equivalent automaton $\mathcal{A} = \{Q, \Sigma, \delta, S, F\}$. When performing the Label or Function Algorithm, each single label or connection point that is inserted in state A is also inserted in B. Since we cannot compute δ by a simple lookup when using ϵ-transitions, we construct yet another equivalent automaton $\mathcal{B} = \{Q', \Sigma, \delta', S', F'\}$ that also incorporates domination with the following algorithm:

- $Q' = Q$.
- δ' is constructed as follows: Every not-ϵ-transition in δ is in δ'. Moreover, when there is a transition from a state q_i to a state q_j and a state q_k which is reachable from q_j only over ϵ-transitions, a transition from q_i to q_k is added.
- $S' = S \cup T$ where T is every state which can be reached from a state $s \in S$ only via ϵ-transitions.
- $F' = F$.

For an example, see Figure 3.4.

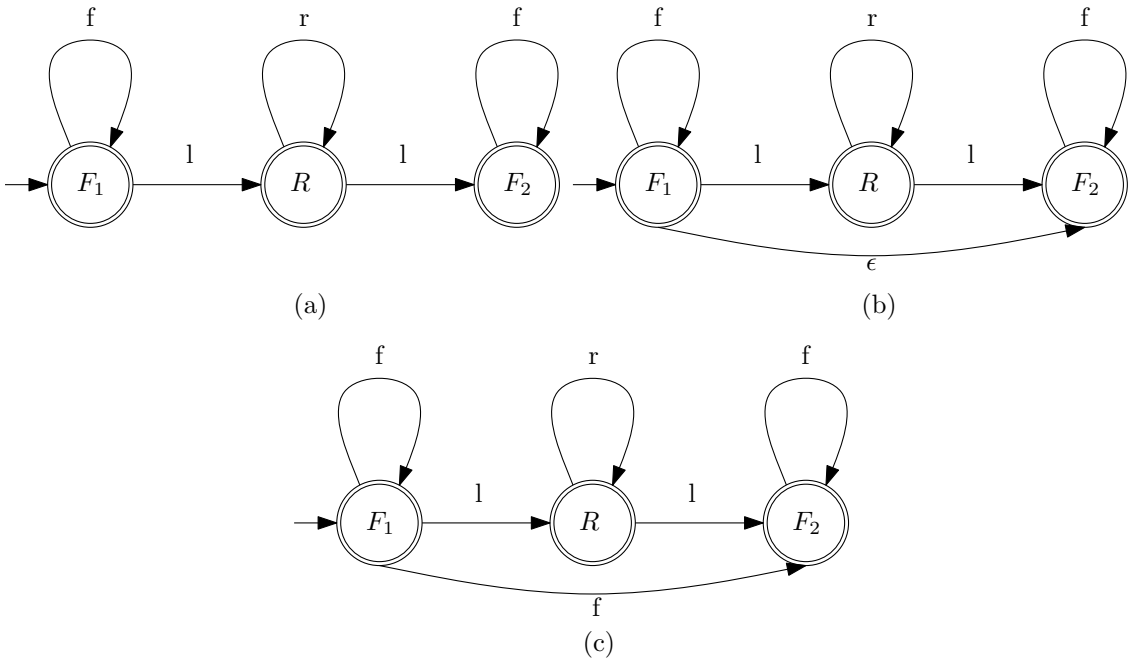


Figure 3.4: The state F_1 in the automaton in Subfigure (a) dominates the state F_2. Therefore, we can add an ϵ-transition from F_1 to F_2 as shown in Subfigure (b). We then construct an equivalent ϵ-free automaton, shown in Subfigure (c).

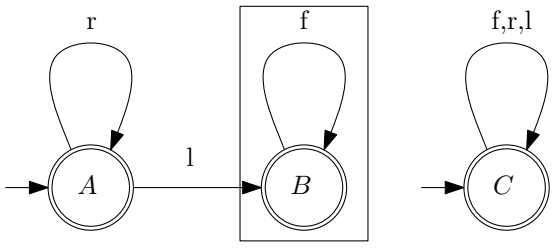


Figure 3.5: Illustration of end-states. State A is no end state because it has transitions to state B. State C is no end state because it has transitions for multiple labels. Only B is an end-state with regard to label f

3.3.2 Backward Search

Propagating whole functions across the network is costly. It seems especially wasteful when propagating them over a time-independent subnetwork, where every single label or function propagated over the same edge increases with the exact same constant. If the target node is in a time-independent subnetwork and the automaton does not allow switching into another subnetwork again, we can accelerate this process. We call such a state that does not allow switching from a subnetwork with label l into another subnetwork an *end-state* with regard to label l. See Figure 3.5 for examples. More formally:

Definition 2. *Given an automaton $\mathcal{A} = \{Q, \Sigma, \delta, S, F\}$, a state $q \in \mathcal{A}$ is an end-state with regard to a label $l \in \Sigma^*$ if and only if $\delta(q, l) = q$, and for every $m \in \Sigma^*$ with $l \neq m$, $\delta(q, m) = \emptyset$ holds.*

If the target node t is in a time-independent subnetwork, we perform a one-to-all query from t to all other nodes on the backward graph of that subnetwork, and store $dist_t(v)$ for each node v. Whenever we relax an edge $e = ((w, q'), (v, q))$ of the product network where v is in the same subnetwork as t and q is an end-state with regard to the label of

the subnetwork, we add $dist_t(v)$ to the label we try to insert at v. We then try to insert this label at t instead. Since we can never leave the subnetwork and t is our target, we do not put t into the priority queue. When applicable, this saves propagating whole functions over this last time-independent subnetwork.

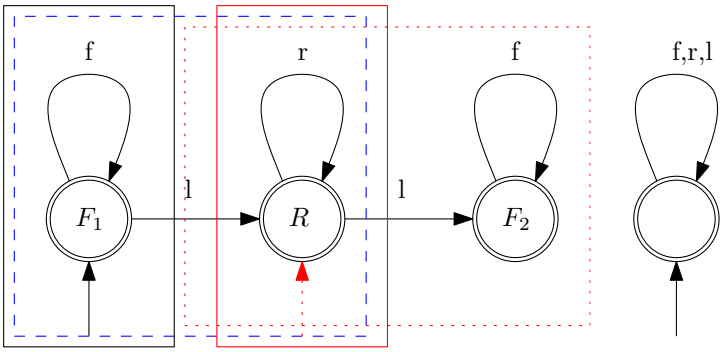
4. Evaluation

We conducted experiments on one core of an Intel Xeon E5430 processor running SUSE Linux 11.1. It has 32 GiB of RAM, a 12 MiB of L2 cache. We compiled the program with GCC 4.5, using optimization level 3. The code is written in C++ and we use the STL. The priority queue is a 4-ary heap.

Input Networks. The networks are those used in [DPW12b], but with the color model applied to the rail networks [DKP10]. More specifically, we perform tests on the `ny-road-rail` and `de-road-rail` instances. The `ny-road-rail` instance consists of New York's foot network and the public transit network operated by MTA. Together, they contain 607 502 nodes, 579 849 for the foot network and 27 203 for the public transit network. The `de-road-rail` instance is made up of the pedestrian and railway networks of Germany. It contains both long and short distance trains. Together, they have 5 075 681 nodes, 5 655 680 for the foot network and 20 001 for the railway networks. Note that public transit network of the `ny-road-rail` instance is much bigger in comparison to its foot network. On the other hand, trains in the `de-road-rail` instance can be much faster, and are therefore more likely a part of a shortest path if we pick source and target at random. We assign the label f to edges of the foot subnetworks, r to edges of the rail subnetworks and l to the link edges.

Input Automata. As automata we use the automata shown in Figure 4.1. In the `road` and `rail` automata only the foot or rail network may be used respectively. In the `road/rail` automaton the rail network may be entered only once or be omitted entirely. In the `road`→`rail` and `rail`→`road` automata the network may be switched exactly once. For all these automatons their states are denoted by F_1 (first foot state), R (rail state) and F_2 (second foot state), as can also be seen in Figure 4.1. There are no restrictions when using the `everything` automaton. Moreover we use the `road/rail [dom]` automaton to allow domination between states for the `road/rail` automaton. The automaton `road/rail [dom]` is the same as the one depicted in Figure 3.4(c).

Methodology. We first perform one-to-all queries, then point-to-point queries and finally point-to-point queries with a backward search for the Function and Label Algorithm where applicable. We compare the Function and Label Algorithm with an *LC-Dijkstra* (see Chapter 2.5) in the following way: For each departure time of the connection points of the function that the Label and Function Algorithm compute, we perform a time query using an *LC-Dijkstra* for that departure time. If the function is constant, we run one time query with an arbitrary departure time. This is like performing an algorithm that knows all



(a) The `road/rail` automaton and its subautomata (b) The `everything` automaton

Figure 4.1: Subfigure (a) shows the `road/rail` automaton, divided in its subautomata. The `road→rail` automaton is confined by the blue dashed lines, the `rail→road` automaton by the red dotted lines. The `road` automaton consists only of F_1, and the `rail` automaton only of R. In Subfigure (b) we see the `everything` automaton that allows all paths.

relevant departure times in advance, and computes the shortest path for each of them. We call this hypothetical algorithm the *PLCD-algorithm* (Profile-Label-Constrained-Dijkstra). Note that when we use the `road/rail [dom]` automaton, we compare this to a PLCD-algorithm using the `road/rail` automaton. This is because we use this automaton only to achieve a speedup for the Function and Label Algorithm, but the automaton leads to a slowdown for normal time queries as we may settle node/state combinations (v, F_2) unnecessarily. For each type of query and each automaton we run 100 queries with each algorithm. When we perform a backward search (this is only applicable for `road/rail` and `rail→road`), we denote this by the suffix `[back]`. When using the `everything` automaton, source and target node are picked uniformly random. For all other automatons, we choose the source and target node as follows: If the initial state of the automaton is F_1, we pick a random node in the foot network as source node. Otherwise, the initial state of the automaton is R, and we pick a random node in the rail network as source node. With exception of the `everything` automaton, each automaton has exactly one end-state with regard to a label. If this label is f, we pick a random node of the foot network as target node. If the label is r, we pick a random node from the rail network as target node. We measure the query time in milliseconds. Furthermore we measure the number of connection points that have been settled. For the PLCD-algorithm this is the amount of settled nodes, for the Function Algorithm the number of settled nodes times the size of their functions, denoted by *settled conns*, and for the Label Algorithm the amount of settled single labels denoted simply by *settled labels*. The size of the computed profile function is denoted by S_t. We show the speedup of the Function and Label Algorithm over the PLCD-algorithm, denoted by *Sp. up*. For the Function Algorithm we furthermore measure the amount of settled nodes and how often each node that is settled at least once in a state is settled on average in this state. For the Label Algorithm we additionally measure the maximal size of a labelset in one run, denoted by *average max labelset size*, and the maximal labelset size in all runs, denoted by *max labelset size*.

In the following we first analyze the query performance and then look at more detailed statistics for the Function and the Label algorithm.

Table 4.1: **Query performance** on the `ny-road-rail` instance. We present one-to-all queries and point-to-point queries. The column denoted $|S_t|$ shows the average size of the computed profile functions. The PLCD column shows the average query time of the hypothetical PLCD-algorithm and its average search space. The Function and Label Algorithm columns show the average query time and amount of settled connections or single labels, respectively. We also report for both these algorithms the speedup in comparison to the PLCD-algorithm.

		PLCD		Function Algorithm			Label Algorithm		
Automaton	$\|S_t\|$	Settled Labels	Time [ms]	Settled Conns	Time [ms]	Sp. up	Settled Labels	Time [ms]	Sp. up
`ny-road-rail` one-to-all									
`road`	1	580 k	282	580 k	496	0.6	580 k	521	0.5
`rail`	68	1 803 k	898	6 267 k	822	1.1	1 818 k	2 326	0.4
`road/rail`	14	15 513 k	8 733	72 880 k	13 736	0.6	11 962 k	19 952	0.4
`road→rail`	33	19 468 k	11 181	3 797 k	1 056	10.6	1 491 k	1 805	6.2
`rail→road`	31	18 222 k	9 221	134 612 k	19 910	0.5	21 922 k	38 998	0.2
`everything`	16	9 215 k	5 305	54 099 k	10 229	0.5	10 594 k	20 079	0.3
`ny-road-rail` point-to-point									
`road`	1	289 k	137	289 k	251	0.5	289 k	263	0.5
`rail`	68	766 k	345	6 210 k	810	0.4	1 775 k	2 277	0.2
`road/rail`	14	5 926 k	3 288	57 428 k	9 661	0.3	6 675 k	13 012	0.3
`road→rail`	33	4 513 k	2 520	3 330 k	669	3.8	1 056 k	1 287	2.0
`rail→road`	31	8 249 k	4 201	126 731 k	18 109	0.2	15 538 k	33 115	0.1
`everything`	16	4 464 k	2 556	41 848 k	7 140	0.4	6 492 k	14 188	0.2

Query Performance

In Table 4.1 and 4.2 we present the query performance of the hypothetical PLCD-algorithm, the Function and the Label Algorithm.

When we perform a query only on the foot network, our algorithms effectively become an *LC-Dijkstra*. In this case, the additional data structures of our algorithms cause a speeddown. The speeddown of the Label algorithm is a bit higher, but not significantly. The number of settled nodes by the Function Algorithms times their function size is always about an order of magnitude higher than the number of settled single labels by the Label Algorithm. The Label Algorithm is faster when using the `rail` or `road/rail` automaton on the `de-road-rail` instance. In all other cases, however, the Function Algorithm is considerably faster than the Label Algorithm. This means that the additional costs for finding the representant and inserting single labels at a node are too high. Moreover, the spatial locality in memory when propagating a whole function is better than extracting each single label individually. The nodes are not settled often enough by the Function Algorithm to outweigh this.

Using a stopping criterion (i. e. performing point-to-point queries) is nowhere as effective for our algorithms as for the PLCD-algorithm. When comparing query times of one-to-all queries with point-to-point queries, the speedup is barely noticeable in some instances. The reason is that we want to compute a Pareto-set of results, and there are a lot more possibilities to improve such a set than just a single value (see Chapter 3.1). There still are significant speedups in some instances, though. When using the `road` automaton, they have the same speedup as the PLCD-algorithm. When using the `road→rail` automaton the speedup is also considerable. The reason is that using trains is faster than walking.

Table 4.2: **Query performance** on the `de-road-rail` instance. We present one-to-all queries and point-to-point queries. The column denoted $|S_t|$ shows the average size of the computed profile functions. The PLCD column shows the average query time of the hypothetical PLCD-algorithm and its average search space. The Function and Label Algorithm columns show the average query time and amount of settled connections or single labels, respectively. We also report for both these algorithms the speedup in comparison to the PLCD-algorithm.

		PLCD		Function Algorithm			Label Algorithm		
Automaton	$\|S_t\|$	Settled Labels	Time [ms]	Settled Conns	Time [ms]	Sp. up	Settled Labels	Time [ms]	Sp. up
`de-road-rail` one-to-all									
`road`	1	5 056 k	2 961	5 056 k	5 083	0.6	5 056 k	5 918	0.5
`rail`	14	267 k	145	944 k	340	0.4	276 k	266	0.5
`road/rail`	18	174 462 k	130 559	453 985 k	118 134	1.1	96 558 k	234 929	0.6
`road→rail`	15	73 293 k	50 019	6 188 k	6 212	8.1	5 374 k	7 051	7.1
`rail→road`	16	78 570 k	61 688	405 989 k	100 438	0.6	85 233 k	206 214	0.3
`everything`	18	86 185 k	74 934	438 238 k	108 719	0.7	93 483 k	240 226	0.3
`de-road-rail` point-to-point									
`road`	1	2 448 k	1 368	2 448 k	2 436	0.6	2 448 k	2 821	0.5
`rail`	14	109 k	52	941 k	337	0.2	267 k	260	0.2
`road/rail`	18	40 212 k	30 074	425 337 k	106 504	0.3	78 709 k	197 189	0.2
`road→rail`	15	1 134 k	692	1 343 k	649	1.1	513 k	634	1.1
`rail→road`	16	36 672 k	27 246	400 822 k	99 669	0.3	76 403 k	184 822	0.1
`everything`	18	38 679 k	31 897	418 001 k	102 945	0.3	81 883 k	212 736	0.1

Since we end in the rail network, we always can reach the destination by train and do not have to walk after that. This is often much faster than walking, so that the stopping criterion holds early relative to the size of the foot network.

Query times can differ significantly for different source/target network combinations. It is striking that in every case where it is likely to enter the foot network after being in the railway network, the query time is higher by an order of magnitude. In case of `de-road-rail`, even two orders of magnitude. This is because we propagate whole functions over the normally time-independent foot network. We can use domination between states and a backward search to avoid this.

In Table 4.3 we present the performance of our improvements. When using an automaton that incorporates domination between states we achieve a speedup over the PLCD-algorithm, though not a significant one. In the `de-road-rail` instance, where trains are much faster in comparison to the `ny-road-rail` instance, the effect is barely noticeable. Still, we only have to adapt the input automaton slightly to achieve this speedup.

When using the `road→rail` automaton or backward searches, our algorithms achieve a considerable speedup over the PLCD-algorithm. This is because in these cases the PLCD-algorithm searches on the time-independent foot network for each departure time, whereas our algorithms do this only once. In the `de-road-rail` instance, we achieve a speedup up to an order of magnitude over the PLCD-algorithm when using backward searches.

Function Algorithm Statistics

In Table 4.4 we present more detailed statistics for the Function Algorithm.

Table 4.3: **Query performance** on `ny-road-rail` and `de-road-rail` with improvements enabled. The column denoted $|S_t|$ shows the average size of the computed profile functions. The Function and Label Algorithm columns show the average query time and amount of settled connections or single labels, respectively. We also report for both these algorithms the speedup in comparison to the PLCD-algorithm.

		Function Algorithm			Label Algorithm		
Automaton [Improv.]	$\|S_t\|$	Settled Conns	Time [ms]	Sp. up	Settled Labels	Time [ms]	Sp. up
Improvements `ny-road-rail`							
`road/rail`	14	57 428 k	9 661	0.3	6 675 k	13 012	0.3
`road/rail [dom]`	14	57 052 k	9 641	0.3	6 642 k	12 913	0.3
`road/rail [back]`	14	2 888 k	950	3.4	917 k	1 464	2.2
`rail→road`	31	126 731 k	18 109	0.2	15 538 k	33 115	0.1
`rail→road [back]`	31	6 218 k	1 319	3.2	1 798 k	2 874	1.5
Improvements `de-road-rail`							
`road/rail`	18	425 337 k	106 504	0.3	78 709 k	197 189	0.2
`road/rail [dom]`	18	424 809 k	106 167	0.3	78 605 k	196 145	0.2
`road/rail [back]`	18	1 233 k	2 965	10.1	425 k	2 903	10.3
`rail→road`	16	400 822 k	99 669	0.3	76 403 k	184 822	0.1
`rail→road [back]`	16	940 k	2 748	9.8	269 k	2 646	10.2

As mentioned in the section above, the number of settled connections of the Function Algorithm is most times significantly higher than the amount of settled single labels of the Label Algorithm. However, the amount of settled nodes of the Function Algorithm is significantly lower than both. When we look at how many times the settled nodes were settled on average, we see that this number increases whenever a new state is entered. In F_1, wherever applicable, all functions are constant and every node that is settled is only settled once. Nodes are settled multiple times when entering the railway network, as this network has non-constant functions. It is remarkable however, that nodes are settled even more often in state F_2. A possible explanation are effects as described in our scenario presented in Figure 3.1. This means the backward search is even more important, since it eliminates the network where nodes are settled the most times. Even so, the nodes are not settled often enough to outweigh the additional costs of the operations performed by the Label Algorithm.

Label Algorithm Statistics

When storing the single labels in a Pareto-set at a node, we would like to reserve memory at the beginning so we do not allocate it at runtime, similar to [DKP10]. In that publication, the number of labels is bound by the number of departures from the source station. In our scenario, this is not the case, as there are no scheduled departures in time-independent networks. In Table 4.5 we depict additional data for the Label Algorithm. The average output size is not large, but we see that the maximal number of labels in a labelset is much higher. While it is feasible to reserve that much memory space for the rail network, the number of nodes in our foot networks are orders of magnitudes higher, and in the worst case we have to store labels for each node/state combination. This is the case when using the `everything` automaton

Table 4.4: **Function Algorithm.** We report a detailed performance analysis of the Function Algorithm for different automatons and our improvements. We show both the amount of settled nodes and settled connections and give a per state breakdown of the search space: For every state of the query automaton we report how often a node is settled on average in that state (over every node settled in that state). For comparison we present the amount of single labels settled by the Label Algorithm in the column Settled Labels. We put the only state of the `everything` automaton into the column of state F_1.

Automaton [Improv.]	Settled Labels	Settled Nodes Function	Settled Conns Function	Average Settled per Node in F_1	Average Settled per Node in R	Average Settled per Node in F_2
`ny-road-rail` point-to-point						
`road`	289 k	289 k	289 k	1.0	—	—
`rail`	1 775 k	89 k	6 210 k	—	2.9	—
`road/rail`	6 675 k	1 746 k	57 428 k	1.0	2.0	5.0
`road/rail [dom]`	6 642 k	1 862 k	57 052 k	1.0	2.0	4.5
`road/rail [back]`	917 k	306 k	2 888 k	1.0	2.0	0.0
`road→rail`	1 056 k	279 k	3 330 k	1.0	2.6	—
`rail→road`	15 538 k	2 374 k	126 731 k	—	2.9	7.3
`rail→road [back]`	1 798 k	89 k	6 218 k	—	2.9	0.0
`everything`	6 492 k	1 340 k	41 848 k	3.5	—	—
`de-road-rail` point-to-point						
`road`	2 448 k	2 448 k	2 448 k	1.0	—	—
`rail`	267 k	64 k	941 k	—	2.8	—
`road/rail`	78 709 k	24 105 k	425 337 k	1.0	3.1	4.5
`road/rail [dom]`	78 605 k	24 100 k	424 809 k	1.0	3.1	4.5
`road/rail [back]`	425 k	213 k	1 233 k	1.0	3.1	0.0
`road→rail`	513 k	299 k	1 343 k	1.0	3.1	—
`rail→road`	76 403 k	23 929 k	400 822 k	—	2.8	4.3
`rail→road [back]`	269 k	64 k	940 k	—	2.8	0.0
`everything`	81 254 k	23 143 k	418 001 k	4.2	—	—

Summary

The Function Algorithm is faster than the Label Algorithm in almost every case. Single labels are only settled once in the Label Algorithm, but the nodes are not settled often enough by the Function Algorithm to outweigh the costs of propagating each single label individually. The biggest problem for both algorithms is that functions or many single labels are propagated over the time-independent foot network, although for a function or group of single labels the traveltimes increase by the same amount over a time-independent edge. This leads to higher query times in order of magnitudes. Where applicable, this can be reversed by a backward search, so we even achieve a speedup over the hypothetical PLCD-algorithm up to an order of magnitude.

Table 4.5: **Label Algorithm.** For each automaton and our improvements we present the average number of single labels settled per query. We analyze how these labels are distributed in detail: We report the average size of the profile function at the target node ($|S_t|$), the average maximum size of a labelset during one query, and the maximal size of a labelset measured over all queries.

Automaton [Improv.]	Settled Labels	$\lvert S_t \rvert$	Average Max Labelset Size	Max Labelset Size
`ny-road-rail` point-to-point				
`road`	289 k	1	1	1
`rail`	1 775 k	68	175	370
`road/rail`	6 675 k	14	134	704
`road/rail [dom]`	6 642 k	14	133	677
`road/rail [back]`	917 k	14	114	469
`road→rail`	1 056 k	33	87	469
`rail→road`	15 538 k	31	209	699
`rail→road [back]`	1 798 k	31	175	370
`everything`	6 492 k	16	88	1 151
`de-road-rail` point-to-point				
`road`	2 448 k	1	1	1
`rail`	267 k	14	49	161
`road/rail`	78 709 k	18	67	327
`road/rail [dom]`	78 605 k	18	64	306
`road/rail [back]`	425 k	18	55	306
`road→rail`	513 k	15	56	306
`rail→road`	76 403 k	16	64	255
`rail→road [back]`	269 k	16	49	161
`everything`	81 254 k	18	68	432

5. Conclusion

We developed, implemented and evaluated two algorithms that compute profile queries on multimodal networks.

The first one is the Function Algorithm, which is label-correcting. It is an adaption of an algorithm that computes profile queries in unimodal networks by propagating whole functions instead of scalar values. We combine this with an approach to search a shortest path in a product network. This product network connects a graph and a finite automaton, which is used to ensure that only admissible paths are computed. As key for the functions we use the lower bound of the functions. This allows for a reasonable stopping criterion. The label-setting property, however, does not hold. Connection points are propagated multiple times, which seems wasteful in some scenarios (see Figure 3.1).

We then developed the Label Algorithm, which uses similar techniques as the Function Algorithm, but is a label-setting algorithm that propagates single labels. A single label consists of the departure and traveltime of a journey. When using the traveltime of a single label as key, the label-setting property holds so each single label is only settled once. We store Pareto-sets of single labels for each combination of a state of the finite automaton and a node of the graph. On these sets we need to perform several operations to manage the single labels. Implementing those operations efficiently is vital to the running time of the Label Algorithm. We therefore explored different techniques like sorting the single labels by departure or traveltime, or tracking the single label with the smallest key in a Pareto-set. Combining tracking the single label with the smallest key and sorting by departure times is the most promising approach.

We applied further improvements to both algorithms. We developed a method that in certain cases allows comparison of paths between different states, so that not Pareto-optimal solutions can be deleted earlier. Moreover, we saw that propagating functions over time-independent subnetworks leads to a slowdown in orders of magnitudes. Performing a backward search in the time-independent part countermands this. To compare this to Dijkstra's Algorithm, we perform an *LC-Dijkstra* for each relevant departure time. Our algorithms then achieve a speedup of up to an order of magnitude.

Our experimental evaluation showed that the number of settled nodes times their function size in the Function Algorithm is significantly higher than the number of settled single labels in the Label Algorithm. Nevertheless, the Function Algorithm is faster than the Label Algorithm in almost every instance. The additional costs for the Label Algorithm to find the representant and to insert a single label are too high. Moreover, the spatial

locality in memory when propagating whole function is better than extracting each single label individually from the priority queue. The nodes are not reinserted often enough by the Function Algorithm to outweigh this.

Outlook. The propagation of non-scalar values over time-independent subnetworks leads to a high slowdown. Therefore, the next step is to minimize this. Our backward search does not work in all cases, an implementation of Contraction Hierarchies for the time-independent part as in [DPW12b] certainly will improve the running time of the algorithms.

For automatons where we visit subnetworks successively, likethe road/rail automaton (see Figure 4.1), exploring a multi-phase approach could be worthwhile. We could perform a complete one-to-all query for a subnetwork, and only then continue onto the following subnetwork. This would mean that we could use different keys for each subnetwork, so advanced techniques like *self-pruning* [DKP10] could be applied. Until the last subnetwork we would not be able to use a stopping criterion, but our experiments show that applying the stopping criterion does not result in a high speedup in most instances anyway.

Finally, we chose the lower bound of a function f as a key. The lower bound of f is determined by the connection point with the smallest departure time. However, when f was settled once, this connection point does not change during the remainder of the algorithm, and does not contribute to further improvements. Since we settle nodes far less often than there are connection points, the same must also be true for a number of other connection points. We could try to identify as many of those connection points as possible, and neglect them when computing the key for the function. If this is possible, we might overcome problematic scenarios as presented in Figure 3.1.

Bibliography

[ADGW10] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks. Technical Report MSR-TR-2010-165, Microsoft Research, 2010.

[Bas09] Hannah Bast. Car or Public Transport – Two Worlds. In Susanne Albers, Helmut Alt, and Stefan Näher, editors, *Efficient Algorithms*, volume 5760 of *Lecture Notes in Computer Science*, pages 355–367. Springer, 2009.

[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.

[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)*, OpenAccess Series in Informatics (OASIcs), pages 209–225, 2007.

[BJM00] Chris Barrett, Riko Jacob, and Madhav V. Marathe. Formal-Language-Constrained Path Problems. *SIAM Journal on Computing*, 30(3):809–837, 2000.

[Dea99] Brian C. Dean. Continuous-Time Dynamic Shortest Path Algorithms. Master's thesis, Massachusetts Institute of Technology, 1999.

[Dij59] Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[DKP10] Daniel Delling, Bastian Katz, and Thomas Pajor. Parallel Computation of Best Connections in Public Transportation Networks. In *24th International Parallel and Distributed Processing Symposium (IPDPS'10)*, pages 1–12. IEEE Computer Society, 2010.

[DPW09] Daniel Delling, Thomas Pajor, and Dorothea Wagner. Accelerating Multi-Modal Route Planning by Access-Nodes. In Amos Fiat and Peter Sanders, editors, *Proceedings of the 17th Annual European Symposium on Algorithms (ESA'09)*, volume 5757 of *Lecture Notes in Computer Science*, pages 587–598. Springer, September 2009.

[DPW12a] Daniel Delling, Thomas Pajor, and Renato F. Werneck. Round-Based Public Transit Routing. In *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*. SIAM, 2012.

[DPW12b] Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. User-Constrained Multi-Modal Route Planning. In *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*. SIAM, 2012.

[DSSW09] 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*, volume 5515 of *Lecture Notes in Computer Science*, pages 117–139. Springer, 2009.

[DW09] Daniel Delling and Dorothea Wagner. Time-Dependent Route Planning. In Ravindra K. Ahuja, Rolf H. Möhring, and Christos Zaroliagis, editors, *Robust and Online Large-Scale Optimization*, volume 5868 of *Lecture Notes in Computer Science*, pages 207–230. Springer, 2009.

[KLC12] Dominik Kirchler, Leo Liberti, and Roberto Wolfler Calvo. A Label Correcting Algorithm for the Shortest Path Problem on a Multi-Modal Route Network. In *Proceedings of the 11th International Symposium on Experimental Algorithms (SEA'12)*, volume 7276 of *Lecture Notes in Computer Science*. Springer, 2012.

[Kle56] Stephen Cole Kleene. Representation of Events in Nerve Nets and Finite Automata. In Claude E. Shannon and John McCarthy, editors, *Automata Studies*, Annals of Mathematics Studies, pages 3–42. Princeton University Press, 1956.

[KLPC11] Dominik Kirchler, Leo Liberti, Thomas Pajor, and Roberto Wolfler Calvo. UniALT for Regular Language Constraint Shortest Paths on a Multi-Modal Transportation Network. In *Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'11)*, volume 20 of *OpenAccess Series in Informatics (OASIcs)*, pages 64–75, 2011.

[Paj09] Thomas Pajor. Multi-Modal Route Planning. Master's thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2009. Online available at `http://i11www.ira.uka.de/extra/publications/p-mmrp-09.pdf`.

[PSWZ08] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient Models for Timetable Information in Public Transportation Systems. *ACM Journal of Experimental Algorithmics*, 12(2.4):1–39, 2008.