Universität Karlsruhe (TH)
Research University • founded 1825

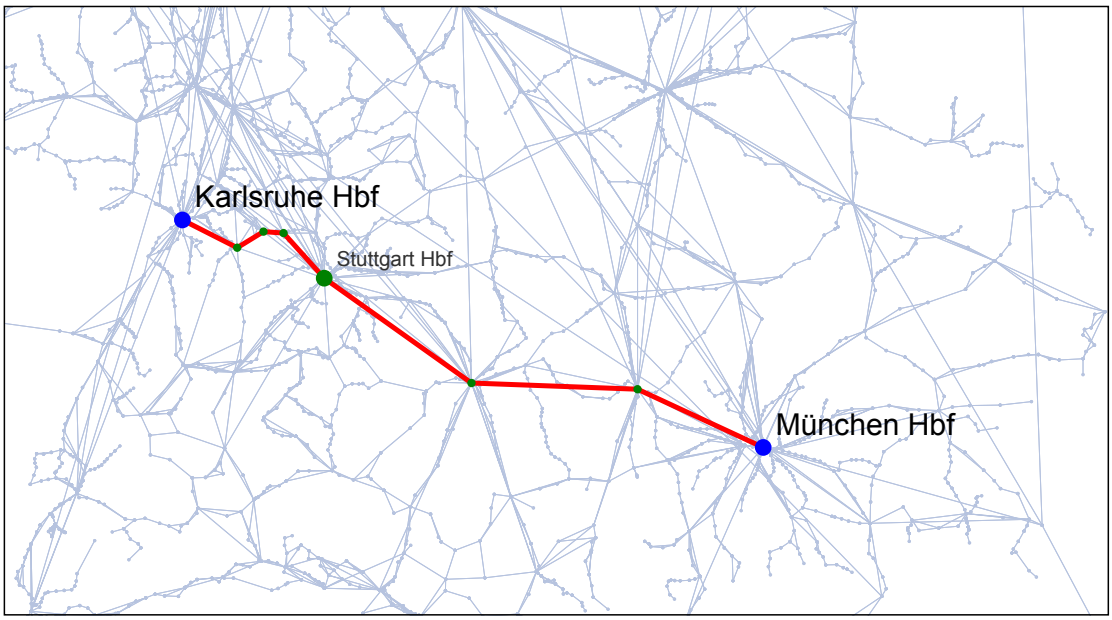
Departement of Computer Science
Algorithmics I • Prof. Dr. Dorothea Wagner

Studienarbeit

Goal Directed Speed-Up Techniques for Shortest Path Queries in Timetable Networks

Thomas Pajor

31st January 2008



Supervisors: Prof. Dr. Dorothea Wagner, Dipl. Inf. Daniel Delling

Abstract

Almost any transport company, like railway companies or air lines, have timetables for planning and scheduling their specific services. The problem of finding an optimal itinerary according to some optimization criterion like travel time can be reduced to a shortest path problem on an adequate graph with non-negative edge weights.

The most prominent algorithm for solving the shortest path problem on such graphs is Dijkstra's algorithm without a doubt. Unfortunately, the data sets are far too huge for a naive implementation of Dijkstra to perform quick enough. Therefore, lots of research has been put into developing speed-up techniques, which give really amazing results on road network graphs. However, on timetable networks the achieved speed-ups, as of today, are far from what can be done on road networks.

Due to time being a crucial aspect on timetable graphs (a road can be used at any time, but a train is only departing at specific times), speed-up techniques specially developed for road networks need to be adapted carefully in order to work properly. Some techniques like highway hierarchies do not work at all, because they require a bidirectional search, which can not be done on timetable networks, since we do not know the arrival time in advance when stating a query.

In this work we present several approaches to model timetable networks to support shortest path queries which minimize travel time. Furthermore, we adapt two unidirectional goal directed speed-up techniques which belong to the best on road networks to work with our time expanded model, namely Arc-Flags and ALT. Both techniques are goal directed in the sense that they try to optimize toward the target station, either geographically or in time. For that reason, we explore the combination of these techniques as well and see that they behave quite orthogonally, as the speed-up of the combined technique multiplies.

Finally, in our experimental study we conduct tests on huge sets of real world data to show the feasibility of our approaches which lead to some very good speed-ups with the combination of the Arc-Flags and the ALT approach.

Contents

1 Introduction

Mobility has gained dramatic importance in modern life. To satisfy these needs huge networks of infrastructure have been built in the past centuries. Road networks on one hand but also public transport networks on the other hand play the most important roles—whether they are railways, buses, planes or ships.

One central problem on any of the networks named above is *route planning*. Imagine living in the south of Germany and having relatives at the North Sea coast. Assuming that we do not know the way around, we have to do some kind of planning before we start our trip. If we plan on going by car, we have to investigate which roads and highways we are going to use. If we prefer riding the train instead, the task is a little bit more complicated. Not only do we need to find out which train lines we have to choose, but we probably also need to switch trains, which involves determining the right connections at transfer stations. Furthermore, trains are only running at specific times and not anytime we arrive at a station. For that reason, departure and arrival times of trains are crucial when planning our journey.

With the German railway network having a length of more than 36,000 km, the task of finding an optimal itinerary for our journey by hand becomes an impossible task. Therefore, many railway companies provide computer aided tools for planning itineraries. For instance, the Deutsche Bahn in Germany uses a system developed by HaCon GmbH [HIm], which is also used in a variety of other countries. Generally they have a central server which holds the data of the train schedules, and clients (e.g. terminals or internet users) state queries to the server. Therefore, answering the queries has to be very efficient in order to handle the vast number of concurrent requests.[1] Commercial tools like HaCon's HAFAS system use heuristics to improve computation speed. Unfortunately, this does not always lead to optimal results regarding travel time or the number of transfer.

In this work we show how periodic timetable information systems[2] can be modeled as directed graphs in such a way, that a shortest path query algorithm can be used to compute an optimal itinerary regarding minimum travel time. The most prominent algorithm for computing shortest paths on a graph with non-negative edge weights is Dijkstra's algorithm [Dij59]. However, realistic graphs are too large for an efficient computation using only plain Dijkstra. Therefore, we study several speed-up techniques which reduce the search space of Dijkstra's algorithm and eventually lead to speed-ups in computation time. We focus on goal directed techniques, since they can be easily adapted to timetable models. Hierarchical techniques like Highway Hierarchies [SS05] require a bidirectional shortest path search which is—as we see later—not possible on our models, hence they can not be used with timetable graphs. However, most recent goal directed techniques, especially SHARC [BD08] and CALT [Sch08] have caught up with the hierarchical techniques and thus are on the same level concerning query times.

This work is organized as follows. In the next section we give some fundamentals about

[1] The server of the German railway network receives around 100 queries per second [Sch05].

[2] A periodic timetable is a timetable which is repeated after some period, e.g. each day or each week.

graphs and timetable networks. Then we give a survey about several approaches how to transform timetable data into graphs suitable for handling shortest path requests which correspond to optimal itineraries. There are two main approaches due to [PSWZ07], namely the *time dependent* and the *time expanded* model. However, apart from giving a description we concentrate on the latter in this work. In Section 4 we introduce the *Earliest Arrival Problem* (EA) which asks for an itinerary with an arrival time as early as possible for a given departure place and time. Based on this we can define the shortest path problem on timetable networks and use DIJKSTRA's algorithm for computing shortest path queries. In Section 5 we are going to introduce several speed-up techniques and show some modifications which need to be done for making them work on our scenario. We concentrate on *unidirectional Arc-Flags* and a method derived from A*, the *unidirectional ALT* algorithm, as well as a combination of these two techniques. Finally, in Section 6 we conduct some experiments on real world data from train and bus schedules of Germany and Europe. Our best technique—a combination of ALT and Arc-Flags—leads to speed-ups up to a factor of 17.4 which is a very good result considering that the shortest path problem on timetable networks is much harder than on road networks [BDW07].

2 Preliminaries

Throughout the whole work we deal with graphs and timetable networks. Therefore, we need to define some basic notation for how we describe *graphs*, *trains*, *stations*, etc. which is introduced in this section.

2.1 Fundamentals of Graph Theory

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

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

A *path* P in G is a sequence of nodes $v_1, v_2, \ldots, v_k$, $k > 1$ such that for each $1 \leq i < k$ the condition $(v_i, v_{i+1}) \in E$ holds. If additionally $v_1 = v_k$, then we call P a *cycle*. Note, that a path may contain certain nodes multiple times without being a cycle. A *subpath* $S \subset P$ is a path itself which is contained in P. We call two nodes $u, v \in V$ *connected*, if there exists a path from u to v. If this is true for all nodes $u, v \in V$, we call the whole graph connected. For a graph G which is not connected, a connected subgraph $G' \subseteq G$ is called a *strong connected component* of G.

Let $w : E \to \mathbb{R}^+$ be a function on the edges of the graph. We interpret $w(e)$ for an

arbitrary edge $e \in E$ as its *weight*. Instead of $w((u,v))$ we write $w(u,v)$ for simplicity. The weight of a path $P = v_1, \ldots, v_k$ on G is the sum of the edge's weights on P, meaning $w(P) := \sum_{i=1}^{k-1} w(v_i, v_{i+1})$.

Throughout our work we mostly do not deal with loops, i.e. our graphs do not contain loops. Also, no multiple edges are considered. Note, that multiple edges are excluded by definition, since we defined E as a set. So if for some reason in an algorithmic step an edge between to nodes u and v should occur multiple times, it is unified into a single edge having the minimum value as its new weight.

2.2 Periodic Timetables and Itineraries

Since we deal with timetable information systems in a mathematical sense, we need a formalization what a timetable actually is. In [PSWZ07] an overview is given and we adopt most of the notation from there. However, since for our purpose timetables are the basis of computing itineraries, we introduce the required terms in this section again. In the previous section we mentioned that timetable information systems can not only be found at railway companies, but also at air lines, ships or other logistics corporations. Although, our notions are mostly based on terms like "stations" and "trains" it should be clear, that these are general concepts which can be transfered to any timetable system to which our definitions can be applied.

A (railway) timetable consists of a set of stations $\mathcal{B}$ and a set of trains $\mathcal{Z}$ which run between stations. Moreover, these trains only travel at certain times. It is satisfying to consider a resolution of one minute, so let $\mathcal{T} := [0, 1439]$ be the set of all natural numbers between 0 and 1439 represented in minutes after midnight (a day has exactly 1440 minutes). For calculating differences between two points in time $t_1, t_2 \in \mathcal{T}$, we define a function $\Delta : \mathcal{T} \times \mathcal{T} \to \mathbb{N}$ by

$$\Delta(t_1, t_2) := \begin{cases} t_2(c) - t_1(c) & \text{if} \quad t_2(c) \geq t_1(c) \\ 1440 - t_1(c) + t_2(c) & \text{otherwise} \end{cases}$$

Please be aware, that Δ is not symmetric, hence the order of the arguments is important. At certain parts in this work it may be more clear if times are expressed in the usual clock format `hh:mm` where `hh` is a two digit number between 0 and 23 representing the hour, and `mm` refers to the minutes. Of course these two representations are equivalent, and it should be obvious how one can convert between them.

An *elementary connection* is a tuple $c := (Z, S_1, S_2, t_d, t_a)$ where c is interpreted as train $Z \in \mathcal{Z}$ running from station $S_1 \in \mathcal{B}$ to station $S_2 \in \mathcal{B}$ departing at t_d from S_1 and arriving at time t_a at S_2. The notion $c(x)$ for an arbitrary field x in the tuple c refers to the value of field x. For example $S_1(c)$ yields the departure station of connection c.

Since we only consider periodic timetables with a period of one day, each connection is valid on every day. To model different *traffic days* we can assign a bit vector of 7 bits to each connection c, which then represents the specific days the train is operating on. In our work we only consider timetables which do not utilize traffic days, so we have no need for an additional bit vector. Please refer to [PSWZ07] for more details concerning traffic days.

Finally, the *length* of a connection, denoted by $\text{length}(c) := \Delta(t_d(c), t_a(c))$, is the time in minutes between $t_d(c)$ and $t_a(c)$. We are now able to derive the formal definition of a timetable from the terms introduced above.

Definition 1 (Timetable). *A traffic timetable is a tuple* $(\mathcal{C}, \mathcal{B}, \mathcal{Z}, \mathcal{T})$ *where* $\mathcal{B}$ *is a set of stations,* $\mathcal{Z}$ *is a set of trains,* $\mathcal{T}$ *points in time (for one day) and* $\mathcal{C}$ *the set of elementary connections the timetable consists of.*

Please note, that one connection $c \in \mathcal{C}$ corresponds to a direct connection of one specific train. In particular, this implies that there are no stopovers between $S_1(c)$ and $S_2(c)$. A train with stops at Karlsruhe, Bruchsal and Stuttgart would resolve to two connections $c_1, c_2 \in \mathcal{C}$ where $S_1(c_1) =$ Karlsruhe, $S_2(c_1) = S_1(c_2) =$ Bruchsal and $S_2(c_2) =$ Stuttgart.

Now, let us consider being at a station where we need to switch trains. Arriving at a certain time t, we most likely are not able to catch any train that leaves right after we just arrived.[3] Therefore, for every station $S \in \mathcal{B}$ we define $\text{transfer}(S)$ to be the *minimum transfer time* at station S. This is the minimal time required to switch trains at station S, which should be chosen large enough to cope with the worst case scenario, e.g. the most far apart platforms.

With the definitions just developed, we are able to define what an itinerary on a certain timetable looks like.

Definition 2 (Itinerary). *An itinerary is a total ordered set* $\mathcal{I} \subseteq \mathcal{C}$ *of cardinality* $|\mathcal{I}|$ *due to a total order relation* $\prec$ *of connections with the property of two subsequent connections* $c_i, c_{i+1} \in \mathcal{I}$ *being valid if and only if* $S_2(c_i) = S_1(c_{i+1})$.

The *length* of an itinerary is the accumulated length of all involved connections plus the time between connections at each station which yields

$$\text{length}(I) := \sum_{i=1}^{|\mathcal{I}|} \text{length}(c_i) + \sum_{i=1}^{|\mathcal{I}|-1} \Delta(t_a(c_i), t_d(c_{i+1})).$$

There are two points which should be mentioned regarding itineraries valid to our definition.

1. Train switching is done implicitly if for two subsequent elementary connections the trains are different. If we respect transfer times, the departure time of the second connection must be far enough away from the arrival time of the first one. We go into more detail regarding this point right away when we discuss the length of an itinerary.

2. Since we do not consider different traffic days (meaning every connection is valid on every day) an itinerary where $t_a(c_i) > t_d(c_{i+1})$ is also considered valid—we would just have to wait $\Delta(t_a(c_i), t_d(c_{i+1}))$ minutes over night at the particular station. This is a simplification over the models presented in [PSWZ07].

[3] Most likely we have to switch platforms.

```
(IR 2269, Karlsruhe Hbf,  Pforzheim Hbf,  10:05, 10:23)
(IR 2269, Pforzheim Hbf,  Mühlacker,      10:25, 10:33)
(IR 2269, Mühlacker,      Vaihingen(Enz), 10:34, 10:40)
(IR 2269, Vaihingen(Enz), Stuttgart Hbf,  10:41, 10:57)
(ICE 791, Stuttgart Hbf,  Ulm Hbf,        11:12, 12:06)
(ICE 791, Ulm Hbf,        Augsburg Hbf,   12:08, 12:47)
(ICE 791, Augsburg Hbf,   München Hbf,    12:49, 13:21)
```

Figure 1: Sample itinerary from Karlsruhe to München based on timetable data from the winter period 2000/2001 in Germany.

If we like to respect transfer times at stations, the length of an itinerary $\mathcal{I}$ needs some extra consideration. The length as defined above is the accumulated length of all connections plus the time we spend at each station. However, it is not enough for the calculation of the station-time of two subsequent connections c_i and c_{i+1} at a station S to simply use $\Delta(t_a(c_i), t_d(c_{i+1}))$, since if $\Delta(t_a(c_i), t_d(c_{i+1})) < \mathsf{transfer}(S)$ we would violate the transfer time criterion. For that reason, we extend the Δ function to operate on two elementary connections by the following definition. Let $c_i, c_{i+1} \in \mathcal{I}$ be two subsequent connections and $S := S_2(c_i) = S_1(c_j)$ the station with its transfer time $T := \mathsf{transfer}(S)$. Furthermore, let $Z_i := Z(c_i)$ and $Z_{i+1} := Z(c_{i+1})$ be the trains of the two connections. Then consider

$$\Delta(c_i, c_j) := \begin{cases} \Delta\big(t_a(c_i), t_d(c_{i+1})\big) & \text{if} \quad Z_i = Z_{i+1} \quad \text{(no transfer time consideration)} \\ \Delta\big(t_a(c_i) + T, t_d(c_{i+1})\big) + T & \text{if} \quad Z_i \neq Z_{i+1} \quad \text{(ensure transfer time holds)} \end{cases}$$

Now we can use the Δ function to completely describe the length of an itinerary with $|\mathcal{I}| > 0$ connections by $\mathsf{length}(I) := \sum_{i=1}^{|\mathcal{I}|} \mathsf{length}\, c_i + \sum_{i=1}^{|\mathcal{I}|-1} \Delta(c_i, c_{i+1})$.

A sample itinerary with one transfer is given in Figure 1. Each line consists of one elementary connection c_i from $\mathcal{I}$. The total length of the itinerary is 296 minutes, if we assume that $\mathsf{transfer}(\text{Stuttgart})$ is less or equal 15 minutes. Going on from here, we introduce graph models in the next section which are able to represent timetables in such a way that a shortest path query results in an optimal itinerary considering its length.

3 Models

In this section we present several approaches how to model timetable systems as directed graphs. When modeling road networks as directed graphs, the following common approach is widely used. Intersections of roads are modeled as nodes, and an edge is inserted between two nodes iff there exists a direct road segment (i.e. with no intersections on it) between the two given intersections[4]. The weights on the edges depend on what we want to optimize. For example, if we want to get fastest paths, we would have something like the average travel time on the specific road segment as weights. Just as well, we could use the length (for example

[4]Most edges would be undirected, but as there might be one-way roads, there are directed edges in the graph.

in kilometers) of the road segment to model distances. As long as no events like predictable traffic jams are taken into account, a shortest path query on such a graph does not depend on the departure time, and hence immediately yields a sequence of road segments we can then use for our journey.

Unfortunately (railway) timetables are a little bit more complex, which does not allow a straight forward adaption of the road network model. This is mainly due to the fact, that a certain segment in the railway network can only be used at specific times, that is to say a train is utilizing it. Furthermore, one segment can have different "travel times". For example between Karlsruhe and Offenburg we can ride a fast train like the German ICE, or use a slow train which takes longer for the same segment in the network.

In the subsequent sections we develop several models how timetable data can be represented as a graph. Starting with the most simple approach, the condensed model we then attend to the time dependent approach giving a short overview about it. The main focus is then on the time expanded approach, where we describe two models, namely the simple and the realistic version which form the basis of all our shortest path queries in this work.

3.1 The Condensed Model

The condensed model is a very basic approach consisting of one graph, namely the *condensed graph* which is essentially a representation of the structure of the railway network consisting of stations and their connections between them. The aspect of departure and arrival time is not yet taken into account with this model.

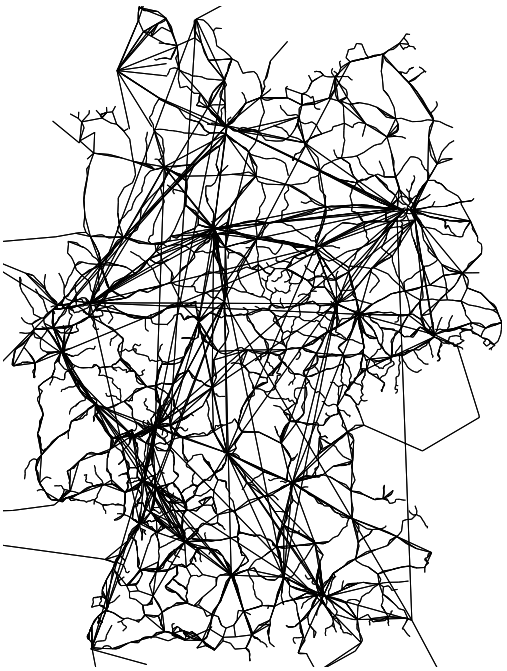
In the condensed graph, denoted by $G^* = (\mathcal{B}, E^*)$, the set of nodes is equivalent to the set of stations, so we use $\mathcal{B}$ as the set of nodes. An edge $e = (S_1, S_2)$ is contained in E^* if there exists at least one elementary connection from S_1 to S_2 in $\mathcal{C}$. For most of our studies there is no need for a weight function in G^*. However, the ALT algorithm later requires us to specify a lower bound between two stations regarding travel time. Hence, we define the weight of an edge $e \in E^*$ to be the best lower bound regarding travel time from S_1 to S_2, i.e. the time the fastest train takes to get along e. Let $\mathcal{C}(e)$ be the set of all elementary connections corresponding to the edge e, then formally the weight $w : E^* \to \mathbb{N}$ is defined by

$$w(e) := \min_{c \in \mathcal{C}(e)} \{\text{length}(c)\}.$$

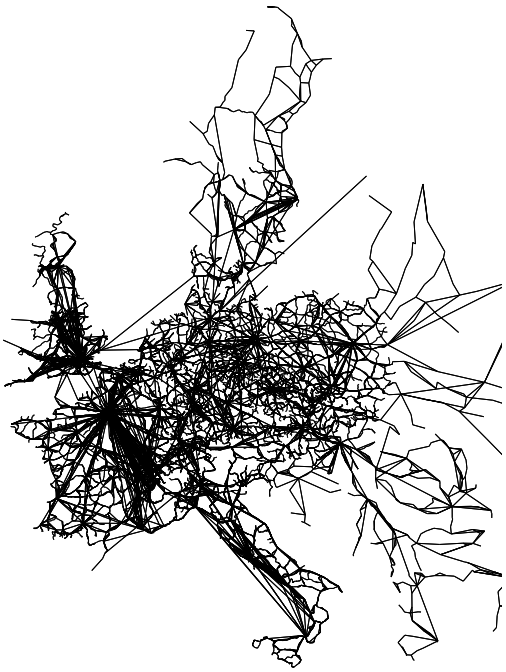
Furthermore, to each node we assign its geographic location in x and y coordinates. So, let $\text{coord}_x : \mathcal{B} \to \mathbb{R}$ and $\text{coord}_y : \mathcal{B} \to \mathbb{R}$ be functions which map a station to its x and y coordinates, respectively. To compute the *distance* between two stations we use the standard Euclidean metric, therefore $\text{dist} : \mathcal{B} \times \mathcal{B} \to \mathbb{R}$ is defined by

$$\text{dist}(S_1, S_2) := \sqrt{\big(\text{coord}_x(S_1) - \text{coord}_x(S_2)\big)^2 + \big(\text{coord}_y(S_1) - \text{coord}_y(S_2)\big)^2}.$$

Of course this model is not sufficient to state realistic shortest path queries, since we have not incorporated any information about the times at which the trains operate.



(a) Germany (2000/2001)



(b) Central Europe (1996/1997)

(c) Berlin's bus network (2000/20001)

Figure 2: Three condensed graphs generated from real world data of the Deutsche Bahn.

```
(ICE 1, Karlsruhe Hbf, Stuttgart Hbf, 10:00, 11:14)
(RE  2, Karlsruhe Hbf, Stuttgart Hbf, 10:05, 11:58)
(ICE 3, Stuttgart Hbf, Karlsruhe Hbf, 10:08, 11:22)
(RE  4, Stuttgart Hbf, Karlsruhe Hbf, 10:26, 12:19)
(ICE 5, Karlsruhe Hbf, Baden-Baden,   10:03, 10:34)
(RB  6, Karlsruhe Hbf, Baden-Baden,   10:11, 11:02)
(ICE 1, Baden-Baden,   Karlsruhe Hbf,  9:07,  9:58)
(RB  7, Baden-Baden,   Stuttgart Hbf,  9:23, 11:42)

transfer(Karlsruhe Hbf) = 6 minutes
transfer(Stuttgart Hbf) = 8 minutes
transfer(Baden-Baden)   = 5 minutes
```

Figure 3: A small fictitious timetable between three stations.

In Figure 2 three examples of condensed graphs are shown. All graphs are based on real world data provided by HaCon [HIm]. Every (straight) line represents an edge in the condensed graph. Since an edge is inserted into the graph as soon as one elementary connection exists in the timetable, we get the effect of having lines crossing through the whole graph, because there might be some trains which have only a few stops and therefore connect stations which are very far apart. The German railway network has 6,730 stations and 19,088 edges; The graph of whole central Europe has 29,770 stations and 91,586 edges and the (relatively small) graph of the local bus network of Berlin has only 2,874 stations with 7,530 edges.

3.2 The Time Dependent Model

The time dependent approach is the canonical enhancement of the condensed model to incorporate the time aspect of a railway timetable. It has been first developed by Brodal and Jacob [BJ04] in 2004. This section only gives a short overview of their approach before we devote ourselves to the time expanded model which is then used throughout this work.

In the *time dependent graph* each station $S \in \mathcal{B}$ again becomes a node, and an edge e between two stations S_1 and S_2 is inserted iff there exists at least one elementary connection from S_1 to S_2. We write $\mathcal{C}(e)$ to describe the set of all elementary connections from S_1 to S_2. The weight of e during the algorithm depends on the time the station S_1 is being considered. Therefore, the weight function $w : E \times T \to \mathbb{N}$ also depends on the set of points in time, and is defined by

$$w(e,t) := \Delta(t, f_{S_1,S_2}(t)), \qquad e := (S_1, S_2)$$

where $f : \mathcal{T} \to \mathcal{T}$ yields the earliest possible time we can arrive at S_2 departing from S_1 at time t (or later).

It should be mentioned that one rather important constraint of this model is, that overtaking of trains between two stations is not allowed, i.e. there must not be two trains Z_1, Z_2 between

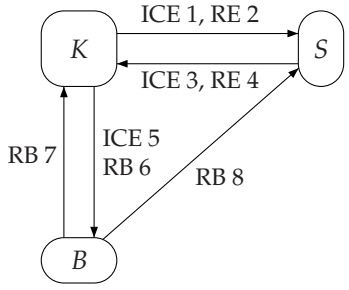


Figure 4: The resulting time dependent graph of the timetable in Figure 3.

two stations S_1 and S_2 such that S_1 leaves at Z_1 first but arrives at S_2 second. If we would allow this feature, the shortest path problem on such networks becomes $\mathcal{NP}$-hard. This has been shown by Orda in [OR90].

Consider the following simple example of a fictitious timetable. We have three Stations: **K**arlsruhe, **B**aden-Baden and **S**tuttgart, with the elementary connections shown in Figure 3. The resulting graph is shown in Figure 4. If we happen to arrive in K at time $t =$ 9:58 and consider the weight on the edge $e = (K, S)$, the function $f_{K,S}(t)$ yields 11:14 and thus $w(e, t) = \Delta(t, f_{K,S}(t)) = 76$ minutes.

The Dijkstra algorithm can now be modified to perform a shortest path query with dynamic weights [CH66]. This can be used to compute itineraries with optimal travel time as proven by Brodal and Jacob in [BJ04].

Since in this work we do not use the time dependent model, we do not go into more detail. A much different approach is the time expanded model. For one timetable our shortest path algorithms operate on two different graphs, namely the *time expanded graph* and the *condensed graph* which we already described in Section 3.1. Since the condensed graph does not hold any information about arrival or departure times, we construct the *time expanded graph* where we "roll out" all the time dependencies in the timetable. For that reason the graph becomes pretty huge, but in exchange Dijkstra's algorithm can be used with much less alteration required than in the time dependent model. Therefore, many speed-up techniques developed for the basic Dijkstra algorithm can be used with the time expanded model as well. Our models are based on those developed in [PSWZ07] and [MHSWZ07]. For historic reasons, there are two versions of the time expanded model, which we introduce both. The *simple model*, like the name says, is a relatively simple approach for modeling timetables. It does not support transfer times, which means we can switch trains in no-time.[5] To encounter that, we enhance it to the *realistic model* which respects transfer times, but as a disadvantage the resulting graph is about 50% larger regarding the number of nodes.

[5] Note: The version of the time dependent model mentioned earlier also does not respect transfer times. However, there is also a "realistic version" of the time dependent model, which is presented in [PSWZ07].

3.3 The Simple Time Expanded Model

The simple model is the most basic approach to expand the time dependencies of the timetable in order to allow shortest path queries. The condensed graph from Section 3.1 is used to represent the structure of the network, while the *time expanded graph* is used to roll out the time dependencies of the timetable.

For that, we consider *events* that occur in our timetable. For the simple model, there are the following two event types:

Arrival event – An arrival event occurs every time a train is arriving at a specific station. For every elementary connection $c \in \mathcal{C}$ there is an arrival event of train $Z(c)$ at station $S_2(c)$ at time $t_a(c)$.

Departure event – A departure event is the counterpart of an arrival event. For every connection $c \in \mathcal{C}$ there is a departure event at station $S_1(c)$ at time $t_d(c)$ of train $Z(c)$.

In the time expanded graph $G = (V, E)$ each event is modeled as a node $v \in V$. This yields two different types of nodes, namely *arrival nodes* and *departure nodes*. We denote $\text{type}(v)$ to be the node type of v, $\text{station}(v)$ to be the station to which the node (event) belongs and $\text{time}(v)$ the time at which the event occurs. The edge set $E = E_c \dot\cup E_t$ is constructed from the subsets E_c, which describes the connections *between* stations, and E_t, the transfer edges *in* stations. For each elementary connection $c \in \mathcal{C}$ there is an edge $e \in E_c$ such that it connects the nodes corresponding to their respective departure and arrival events. Formally, this means an edge $e = (u, v)$ is contained in E_c if and only if there is an elementary connection $c \in \mathcal{C}$ such that

1. u is the node corresponding to the departure event consisting of $Z(c)$, $S_1(c)$ and $t_d(c)$,
2. v is the node corresponding to the arrival event consisting of $Z(c)$, $S_2(c)$ and $t_a(c)$.

The set of transfer edges E_t is constructed as follows. For each station $S \in \mathcal{B}$ let $V(S)$ be the set of all nodes with $\text{station}(v) = S$. Consider the nodes in $V(S)$ ordered such that for two nodes $u, v \in V(S)$ the relation $u \leq v$ holds if and only if $\text{time}(u) \leq \text{time}(v)$. Then E_t is the transitive reduction of the partial order relation $\leq$. To allow transfers over night, we add an additional edge from the latest node in $V(S)$ to the earliest node in $V(S)$ to E_t .

The edge weight function on G is always equal to the distance in time between the respective nodes. Since each node v has its own timestamp $\text{time}(v)$, edge weights $w : E \to \mathbb{N}$ are simply defined by

$$w(e) := \Delta(\text{time}(u), \text{time}(v)), \qquad e = (u, v).$$

Figure 5 illustrates the structure of the simple time expanded graph for two stations of our sample timetable. As we see, the size of the time expanded graph is significantly larger than the size of the condensed graph. This is due to the fact that for real world timetables we

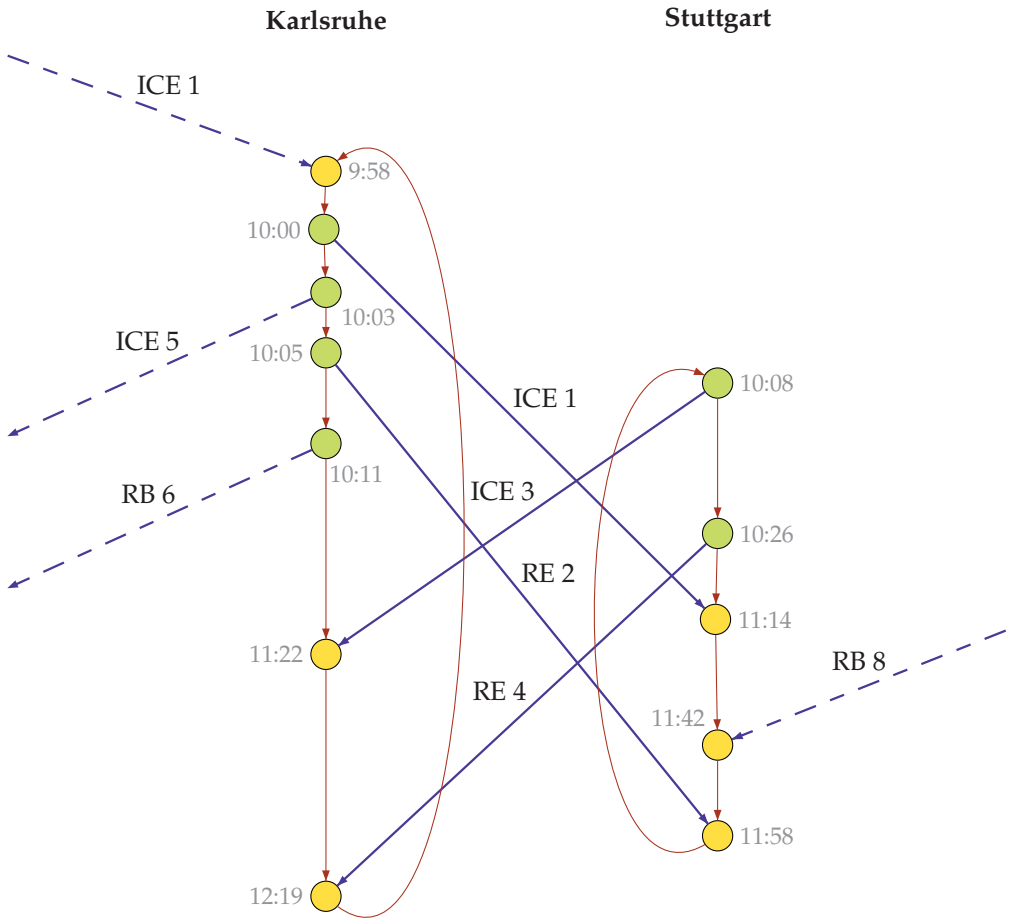


Figure 5: Small sample of two stations in the simple time expanded model. Arrival nodes are yellow and departure nodes are marked green. To each node (event) a timestamp is assigned. Considering the edges there are blue edges, which we can think of the actual elementary connections from the timetable. Furthermore, the red edges allow switching trains in the station.

have numerous trains running between two station S_1 and S_2. Whereas in the condensed graph (and in the time dependent graph as well) there would be only one edge to model this, we get—considering we have k connections—k edges for the connections and also $2k$ nodes. Furthermore, we get another $2k$ edges representing the transfer edges.

The biggest disadvantage of the simple model is that it does not account for transfer times. If we look at Figure 5, we see that we could arrive in Karlsruhe at 9:58 and almost immediately depart at 10:03 with a different train. Since this is quite unrealistic in real world scenarios, we already introduced transfer times in Section 2.2. We now enhance the simple model by this concept in the next section.

3.4 The Realistic Time Expanded Model

The realistic model is based on the simple model but is extended in such a way, that transfer times at stations are respected.

Again, we use arrival and departure events to build our node set V, however, there are three types of nodes now:

Arrival nodes – For each arrival event there exists an arrival node in the graph.

Departure nodes – For each departure event we construct one departure node in the graph.

Transfer nodes – Additionally to the departure node, we add another node type, namely the transfer node. So, for each departure event in the timetable there is a pair of nodes consisting of a transfer node and a departure node.

Let again $\text{type}(v)$ denote the type of node v, $\text{time}(v)$ its time[6], and $\text{station}(v)$ the station it belongs to. Furthermore, let $V(S)$ be the set of all nodes $v \in V$ where $\text{station}(v) = S$. The reason for adding a transfer node for each departure event is, that we want to prohibit paths $u, \ldots, v$ in the graph where u is an arrival node, v is a departure node belonging to the same station S as u, and $\Delta(\text{time}(u), \text{time}(v)) < \text{transfer}(S)$.

For that, the edge set E is made up a little bit more complicated. To make things more clear, we derive the edge set incrementally.

1. Each pair of transfer and departure nodes is connected by an edge $e = (u, v)$, where u is the transfer and v the departure node. This has the semantics of boarding a new train.[7]

2. For each train Z that does not begin or end in S, we need an edge which symbolizes staying in the train (meaning we do not get off the train at S). Therefore we insert a *train edge* between the arrival node belonging to Z at S and the departure node at S belonging to the same train Z.

3. From each arrival node at S we insert an edge to the earliest transfer node respecting the transfer time at S. Formally, this means, if u is an arrival node at S and $t = \text{time}(u)$ the time of the arrival event, an edge $e = (u, v)$ is inserted with v defined by

$$v := \underset{v \in V(S)}{\text{argmin}} \left\{ \Delta\big(\text{time}(u) + \text{transfer}(S), \text{time}(v) \big) \right\}$$

   This can be seen as getting off the train at S.

[6] Note: Two nodes u, v which belong to the same departure event have the same timestamp, i.e. $\text{time}(u) = \text{time}(v)$.

[7] At a station S in the simple model we did not make a difference between boarding a new train or staying in the same train. However, we have to make this difference now, because it should be allowed to leave S with the same train even though the difference between arrival and departure time is less than $\text{transfer}(S)$.

4. Each elementary connection has its edge in the graph exactly the same way as in the simple model. If $c \in \mathcal{C}$ is an elementary connection, then there is an edge from u to v where u is the departure node belonging to the train $Z(c)$ at the departure time $t_d(c)$ at $S_1(c)$. The node v is the arrival node at $S_2(c)$ at $t_a(c)$ with the same train $Z(c)$.

5. Let $\mathsf{Trans}(S) \subseteq V(S)$ be the set of all transfer nodes belonging to a station S. To allow transfers we connect the nodes from $\mathsf{Trans}(S)$ the same way we connected the nodes in the simple model. So, let's consider $\mathsf{Trans}(S)$ ordered with two nodes $u, v \in \mathsf{Trans}(S)$ being in relation $u \leq v$ if and only if $\mathsf{time}(u) \leq \mathsf{time}(v)$. Then again the transitive reduction of $\leq$ is added to E. For allowing transfers over midnight we add an additional overnight edge from the latest to the earliest node in $\mathsf{Trans}(S)$.

Finally, the edge weight function $w : E \to \mathbb{N}$ is, again, the difference in time of the nodes the edge connects, meaning

$$w(e) := \Delta(\mathsf{time}(u), \mathsf{time}(v)).$$

Please note, that every edge between the pairs of transfer and departure nodes has an edge weight of zero, since these nodes share the same timestamp. This has to be taken some extra care of when implementing Dijkstra's algorithm. In general, we can not rule out zero weights—even in the simple model—since it is well possible that more than one train departs at the same time, and therefore the weight of the edge connecting these events would be zero in any case. Furthermore, when we look at bus timetables it is possible that the stations are so close by, that the length of the elementary connection, and hence the edge weight belonging to this connection, could become zero as well.[8]

Figure 6 shows a small excerpt of our example timetable from Figure 3. A few incoming connections at Karlsruhe have been left out to increase readability. As we can see, it is no longer possible to transfer from the arriving ICE 1 at 9:58 to the departing ICE 5 at 10:03, since we assume $\mathsf{transfer}(K)$ to be greater than five minutes. However, it is possible to stay in the ICE 1 and depart to Stuttgart at 10:00.

The size of the expanded graph is even larger than in the simple model, since—assuming we have k elementary connections in $\mathcal{C}$—for $|V|$we get $3k$ nodes, and for $|E|$ we even get k edges for the actual connections, another k edges connecting the transfer with the departure nodes, k edges again between the transfer nodes and k edges connecting each arrival node to one of the transfer nodes. Further, we get $k' \approx O(k)$ edges connecting the arrival with the respective departure node of non ending trains.[9] All together this yields $|E| = 4k + O(k) \approx 5k$ edges.

3.5 Foot Edges

The realistic model gives us a pretty good basis for computing itineraries on timetables. However, there is yet another common problem we have to take care of. If we look at Figure 7

[8] Although we could avoid this through increasing the resolution in time.

[9] We can assume $k' \approx O(k)$ since most of the incoming trains at a specific station also leave the station.

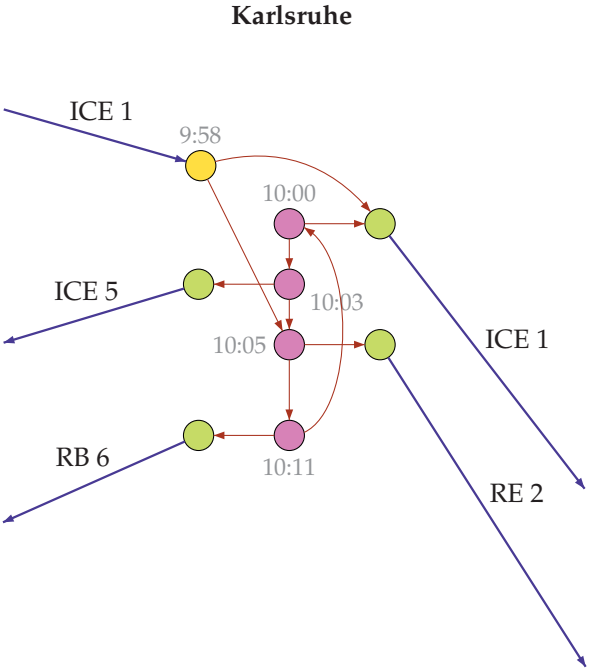


Figure 6: A small portion of our exemplary timetable in the realistic model. Yellow nodes are arrival nodes, the purple nodes are transfer nodes and the green nodes are their associated departure nodes. Edges belonging to elementary connections of the timetable are marked blue, while edges modeling transfers are shown in red. For clarity the timestamp of the departure nodes is omnitted here—in fact they have the same timestamp as their corresponding transfer nodes.

we see a detail of the condensed graph of Germany. If we arrive at Karlsruhe's main station and want to switch to the "Stadtbahn" we have to go to the square in front of the main station since the Stadtbahn stop is located there. Formally the stations "Karlsruhe Hbf." and "KA Bahnhofsvorplatz" are two different stations, and therefore switching trains is not possible here, since there are no connecting edges between these two stations in the graph. In fact, the network of the Stadtbahn is disjoint from the rest of the railway network in the condensed graph, and hence there is no way of computing shortest paths between stations in the Stadtbahn network and the rest of Germany.

To counteract this problem, foot connections are introduced. These are basically additional edges (foot edges) that are added to the realistic time expanded graph to model the act of walking by foot between two stations. For that, we say that two stations $S_1, S_2 \in \mathcal{B}$ are called *neighbours* if for some reason a foot connection should be inserted between them. In principle, there is no general rule as to which two stations should become neighbours. Normally, one would choose these neighbouring stations by hand when constructing the timetable. Since our real world data does not contain any information on this issue, we choose the following rules as to which two stations S_1 and S_2 become neighbours.

- The geographical distance between S_1 and S_2 has to be less than a fixed value $d \in \mathbb{R}$.

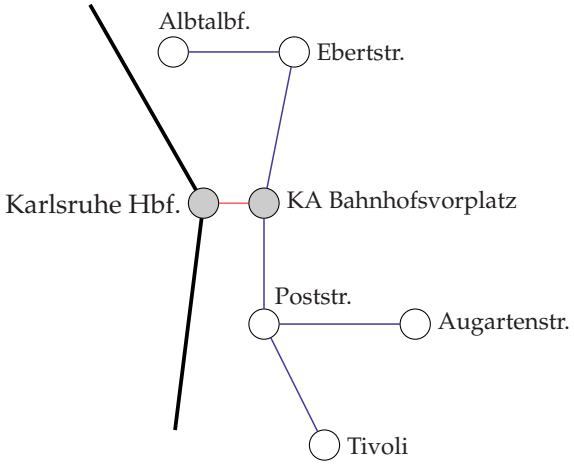


Figure 7: A detail of the condensed graph of Germany around Karlsruhe main station. Without the red edge the network of the local stadtbahn in Karlsruhe would be disjoint from the rest of the regular railway network, and therefore no shortest paths would exist between these two subnetworks.

- There must not exist any elementary connection between S_1 and S_2 or vice versa. This rule is due to the fact, that in dense bus networks it is well possible that certain bus stops are closer to each other than d, and thus would become neighbours even though a bus connection exists between them (We always want to prefer riding by any means of transportation rather than walking by foot).

Now, let $N \subseteq \mathcal{B} \times \mathcal{B}$ denote the set of chosen neighbour pairs. For a tuple $(S_1, S_2) \in N$ we want foot edges to be inserted from S_1 to S_2. In general it makes sense having N to be a reflexive relation, i.e. if $(S_1, S_2) \in N$ then also $(S_2, S_1) \in N$. Further, let σ be the speed at which one can assume the average person travels by foot. Then the edge set E of the expanded graph G is enhanced by the following edges. For each pair $(S_1, S_2) \in N$ let $\mathrm{Arr}(S_1) \subseteq V(S_1)$ denote the set of all arrival nodes at station S_1 and $\mathrm{Trans}(S_2) \subseteq V(S_2)$ the set of all transfer nodes at station S_2. Then for each node $u \in \mathrm{Arr}(S_1)$ a foot edge $e = (u, v)$ is inserted toward the earliest node from $\mathrm{Trans}(S_2)$ which can be reached in time by foot. Formally the node v is selected by

$$v := \operatorname*{argmin}_{v \in \mathrm{Trans}(S_2)} \Big\{ \Delta\big(\mathrm{time}(u) + \mathrm{dist}(S_1, S_2)/\sigma + \xi, \mathrm{time}(v) \big) \Big\}.$$

The term ξ is a constant that is added to the raw foot travel time from S_1 to S_2. For example, we could set it to $\mathrm{transfer}(S_1)$ or $\mathrm{transfer}(S_2)$ or some other constant value.[10] Finally, the edge weight of each foot edge is set to the difference in time between the two connecting nodes, as usual.

[10] In a big station getting off the platform and getting around takes time, though we do not really cover any distance by doing that. In that case the time computed by $\mathrm{dist}(S_1, S_2)/\sigma$ may not be enough to get from S_1 to S_2. In our experiments we have set ξ to zero, though.

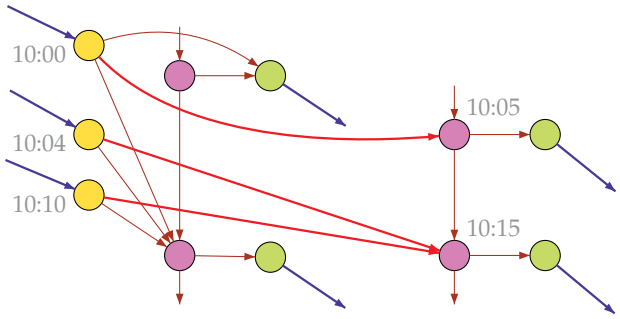


Figure 8: This is a fictional example of two stations connected by foot edges. Assuming a distance of 100 m between the two stations and $d = 3.6$ km/h (which equals 1 m/s) we get a travel time of 100 sec. Each arrival node on the left station is connected with the earliest transfer node to the right (see the bright red edges) which can be reached within 100 seconds from the arrival time.

Figure 8 shows an example how foot edges are inserted. For not making the picture to bloated, only a small part of the expanded graph is shown.

In this section we developed the models which form the basis of our shortest path queries. Thus, the next sections are devoted to the shortest path problem and the speed-up techniques where we see how shortest path queries are connected to optimal itineraries and how query times can be accelerated through the presented goal directed speed-up techniques.

4 Shortest Path Queries

This and the next section are the main sections of our work. We formally introduce the problem of finding an optimal itinerary. Then we present several algorithms which solve the problem, beginning with the basic algorithm of Dijkstra. We see that unlike in road networks there is a tremendous number of different shortest paths for one query. This leads to some strategies for choosing the "best" path along the query. Two possible strategies are then discussed in section 4.4 before we introduce the speed-up techniques in Section 5.

Defining the problem of finding a "good" itinerary may not be as trivial as one might think at first glance. Imagine planning a (rather long) trip by train. One could think of many different optimization criteria through which the itinerary should be computed. For example

a) minimize travel time,

b) minimize transfers—no matter if the journey may take longer thereby,

c) minimize travel distance. Don't take a longer route (even if it may be faster).

d) Consider only some well specified train classes. For example do not choose express trains, since they are more expensive in most countries than slow trains.

e) Minimize costs, e.g. prefer a route which may be longer but is cheaper in return.

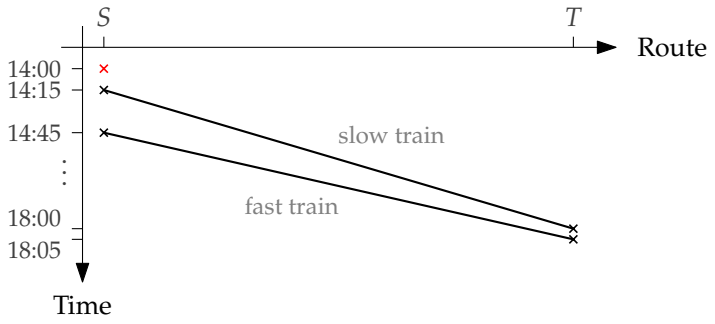


Figure 9: Visualization of two trains departing at S towards T. The slow train takes 25 minutes longer than the fast train, but arrives at T five minutes earlier. Considering pure travel time the fast train should be preferred. However, if we consider the whole time from 14:00 on, the slow train may be the better choice.

f) Combine all of the above possibilities in a well balanced manner.

We primarily concentrate on optimizing travel time. However, thinking further, even this statement does not make entirely clear what is meant. Mostly, we would state a query to some itinerary planning system like this:

> "Give me an itinerary from station S to station T. I want to depart around 14:00 in the noon and minimize the travel time."

Most likely there is no train at exactly 14:00 departing in our direction. Now imagine there are two choices. The first itinerary chooses a train to depart at 14:15 which arrives at T at 18:00. This is an optimal itinerary, as there is no other itinerary which arrives at T earlier. But there might be an alternative connection starting much later at S, for example at 14:45. But taking this train (which might be a faster train), we arrive at T only slightly after the prior train at 18:05. The second version clearly has a shorter time of travel, namely 3 hours and 20 minutes versus 3 hours 45 minutes. But, since we stated, that we want to depart at 14:00 the overall travel time is five minutes longer. Figure 9 illustrates this context. Which of the two connections should be preferred? We clear up this question for our work after introducing the shortest path problem on directed graphs.

4.1 The Shortest Path Problem

The shortest path problem is a classic problem in graph theory [Dij59] and much attention has been put to it, since it is of great importance in many cases. Given a (directed) graph $G = (V, E)$ with an edge weight function $w : V \to \mathbb{R}$, a *shortest path* from a source node $s \in V$ to some target node $t \in V$ is a path $P = v_1, \ldots, v_k$ in G where $v_1 = s$, $v_k = t$ and for any other path $P' = s, \ldots, t$ the inequality $w(P) \leq w(P')$ holds. The shortest path between two nodes does not necessarily have to be unique. The *distance* of a node $v \in V$ from the source node s is the length of any shortest path from s to v, and is denoted by $\text{dist}_s(v)$. For

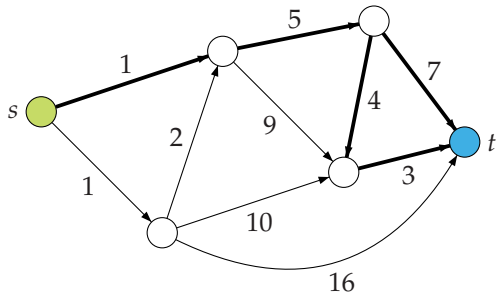


Figure 10: A directed, weighted graph $G = (V, E)$ with two shortest paths between s and t.

simplicity we will only write $\text{dist}(v)$, if it is clear that we mean the distance from the source node.

There are three major version of the shortest path problem [HNR68], each given a directed graph $G = (V, E)$ with some weight function w.

1. The *one-to-one* shortest path problem asks for a shortest path from some source node s to one target node t.[11]
2. The *one-to-many* shortest path problem asks for a set of shortest paths from some source node s to a set $T \subseteq V$ of target nodes.
3. The *one-to-all* shortest path problem is a special case of the one-to-many problem having $T = V$.

Figure 10 shows an example graph with two shortest paths. Next, we define one version of finding an optimal itinerary as a formal problem and reduce it to finding a shortest path.

4.2 The Earliest Arrival Problem

As mentioned before, there are many ways as to how one could refer to an itinerary as being "optimal". In this work we focus on the *earliest arrival problem* (EAP), which is a version of optimizing travel time. The problem has been stated in [PSWZ07] and is as follows.

Problem 1. *Given a source station S and a target station T, as well as a departure time τ_S, find an itinerary with earliest possible arrival time τ_T at T.*

This version of minimizing time prefers the slow train of Figure 9 over the fast one, because the slow train is arriving at our destination first. As we see later, we have a few extra liberties of optimizing second order criteria—like the number of transfers—within the margin of not getting worse regarding time.

The following theorem states, that the earliest arrival problem can be reduced to the problem of finding a shortest path on our time expanded models.

[11] While finding just one arbitrary shortest path can be solved efficiently, the problem of finding all shortest paths between two nodes is much harder.

Theorem 1. *Given the time expanded graph $G = (V, E)$ (of the simple or realistic model), a source station S, a target station T and a departure time τ, the earliest arrival problem can be solved by finding a shortest path from s to t, where*

(i) s is the smallest node from $\mathrm{Trans}(S)$ *with* $\mathrm{time}(S) \geq \tau$. *If no such nodes exists, then*

$$s = \operatorname*{argmin}_{v \in \mathrm{Trans}(S)} \{\mathrm{time}(v)\}.$$

(ii) t is an arrival node from $V(T)$ with minimal distance to s.

Proof. We show the correctness of theorem 1 in two steps. First, we show that a shortest path leads to an optimal itinerary regarding the earliest arrival problem on our timetable. Second, we show that to each valid and optimal itinerary regarding the EAP, a shortest path in G exists. Thus, we have proven the equivalence between shortest paths in our model and optimal itineraries.

Let $P = v_1, \ldots, v_k$ be a shortest path starting at some transfer node v_1 leading to some arrival node v_k. By construction of the expanded graph each edge $e = (u, v)$ where $\mathrm{type}(u) =$ `Departure Node` and $\mathrm{type}(v) =$ `Arrival Node` can be mapped to an elementary connection c in our timetable. To simplify matters, we call these types of edges *connection edges*. Now let $\mathcal{I} = c_1, \ldots, c_{|\mathcal{I}|}$ be the sequence of elementary connections we get by walking along the path P from v_1 to v_k and adding the corresponding elementary connection to $\mathcal{I}$ each time we encounter a connection edge. Since a connection edge has only been inserted for elementary connections from $\mathcal{C}$, the only edges $(u, v) \in E$ with $\mathrm{station}(u) \neq \mathrm{station}(v)$ are either connection edges or foot edges. Hence, for two subsequent elementary connections $c_i, c_{i+1} \in \mathcal{I}$ either $S := S_2(c_i) = S_1(c_{i+1})$ (meaning we transfer at S or stay in our train), or $S_2(c_i) \neq S_1(c_{i+1})$ and thus a foot edge has been used to get from $S_2(c_i)$ to $S_1(c_{i+1})$. In that case these two stations are neighbours in the timetable.

Furthermore, we show that the length of the shortest path equals the length of the itinerary plus the time from the first (transfer) node v_1 to the first departure node on P which we abbreviate by ζ. Formally we show

$$w(P) = \mathrm{length}(\mathcal{I}) + \underbrace{\Delta\big(\mathrm{time}(v_1), \mathrm{time}(v_d)\big)}_{=:\zeta}$$

where v_d is the departure node corresponding to $c_1 \in \mathcal{I}$. This is valid due to the edge weight function in G having exactly the time difference between its nodes by construction. Therefore we get

$$\begin{aligned}
\mathrm{length}(\mathcal{I}) + \zeta &= \sum_{i=1}^{|\mathcal{I}|} \mathrm{length}(c_i) + \sum_{i=1}^{|\mathcal{I}|-1} \Delta(t_a(c_i), t_d(c_{i+1})) + \zeta && (1)\\
&= \sum_{i=1}^{|\mathcal{I}|} w(u_i, v_i) + \sum_{i=1}^{|\mathcal{I}|-1} w(P_{v_i, u_{i+1}}) + \zeta && (2)\\
&= w(P), && (3)
\end{aligned}$$

where the edge (u_i, v_i) is the i'th connection edge on P, and $P_{v_i,u_{i+1}}$ is the subpath of P between v_i and u_{i+1}.

Finally, assume $\mathcal{I}$ is not optimal, meaning there is an itinerary $\mathcal{I}'$ with $\text{length}(\mathcal{I}') < \text{length}(\mathcal{I})$. Since the length of $\mathcal{I}$ and $\mathcal{I}'$ is not identical, the itineraries themselves must be different as well. So let $c_i \neq c'_i$ be the first elementary connection which differs in $\mathcal{I}$ from the one in $\mathcal{I}'$. Furthermore, let v_d, v'_d be the departure nodes corresponding to c_i and c'_i, v_a be the (common) arrival node corresponding to $c_{i-1} = c'_{i-1}$. Since each edge weight is exactly the time difference of its two nodes, the only reason for not going over v'_d is, that no path from v_a to v'_d exists in the graph. This is a contradiction to the construction rules of G since each arrival node is either connected to a transfer node at $\text{station}(v_d)$, if $\text{station}(v_d) = \text{station}(v'_d)$, or—because $\mathcal{I}'$ is assumed to be a valid itinerary—there is a foot edge from each arrival node at $\text{station}(v_d)$ to a transfer node at $\text{station}(v'_d)$ if $\text{station}(v_d) \neq \text{station}(v'_d)$.

The backward direction can be shown analogously. Assume an optimal itinerary $\mathcal{I} = c_1, \ldots, c_{|\mathcal{I}|}$. We can construct a shortest path in G beginning by the transfer node corresponding to the departure event of c_1. For each elementary connection c_i we walk along the connection edge in G and for two subsequent connections c_i, c_{i+1} we create the path according to the following three rules.

- If $S_2(c_i) \neq S_1(c_{i+1})$ there exists a foot edge from the arrival node corresponding to c_i to some transfer node at c_{i+1}. Now walk along the transfer nodes until we reach the transfer node matching the departure event of c_{i+1}.
- If $S_2(c_i) = S_1(c_{i+1})$ and $Z(c_i) = Z(c_{i+1})$ there is an edge from the arrival node of c_i to the departure node of c_{i+1}.
- Otherwise there is an edge from the arrival node of c_i to some transfer node of the current station. Again, walk along the transfer nodes until we reach the transfer node matching the departure event of c_{i+1}.

The length of this constructed shortest path $P = v_1, \ldots, v_k$ equals to the length of $\mathcal{I}$ according to the equations (1)–(3) when setting $\zeta := 0$. Assuming there is a path P' from v_1 to v_k with $w(P') < w(P)$ we could construct an itinerary $\mathcal{I}'$ according to the first part of this proof with length $\text{length}(\mathcal{I}') < \text{length}(\mathcal{I})$ which is a contradiction to $\mathcal{I}$ being optimal. □

The proof only refers to the realistic version of the time expanded model. But it can be easily adapted to cope with the simple model—without the transfer time criterion, of course.

When applying an EAP query, the target node t is not known in advance. This is due to the fact that, as our theorem states, the choice of t depends on the distances of all arrival nodes from the source node s at the target station. So it might seem as if we would need to do a one-to-many shortest path query to the set of all arrival nodes of our target station, and determine the correct arrival node afterward. However, as we see in the next section DIJKSTRA's algorithm can be modified slightly to encounter this problem, and a (modified) one-to-one shortest path query is sufficient.

4.3 Basic Dijkstra

The algorithm of DIJKSTRA is the most prominent algorithm for solving the shortest path problem. It has been developed in 1959 by E.W. Dijkstra [Dij59] and can be used to solve the one-to-many/all and the one-to-one shortest path problems. One limitation of DIJKSTRA's algorithm is that it only operates on (directed) graphs with non-negative edge weights. However, this is not a problem, since our graphs have edge weights ≥ 0 by construction. A description of the basic version of DIJKSTRA's algorithm is shown in Algorithm 1.

Algorithm 1: DIJKSTRA's algorithm

Data: A directed, weighted graph $G = (V, E)$ with all edge weights being non-negative. Further, source and destination nodes $s, t \in V$.
Result: A shortest path from s to t.

```
1  Q ← a priority queue of nodes
2  Q.insert(s, 0)
3  while not Q.isEmpty() do
4      v ← Q.dequeue()
5      if v = t then
           // shortest path found
6          stop
7      forall outgoing edges e = (v, w) do
8          if w is a new node then
9              Q.insert(w, dist(v) + w(e))
10             pre(w) ← v
11         else
12             if dist(v) + w(e) < dist(w) or ( dist(v) + w(e) = dist(w) and
               decideSameKey(v, w) ) then
13                 Q.decreaseKey(w, dist(v) + w(e))
14                 pre(w) ← v
   // no shortest path found
15 stop
```

The procedure is as follows. We use a priority queue Q as data structure throughout the algorithm, which consists of nodes. As key value, the (current) distance of the specific node from s is used. We start by inserting the source node, which has distance 0. Now, as long as Q is not empty, the first node v is dequeued (the one with minimal distance from s), and all outgoing edges of v are being iterated over. For each node w that can be reached through v, we either add w to the priority queue (if w has not been touched yet), or if the distance of

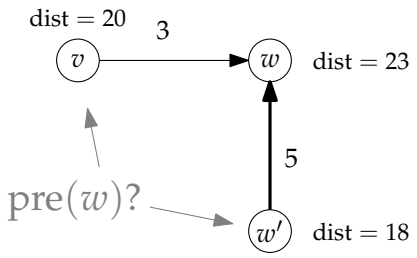


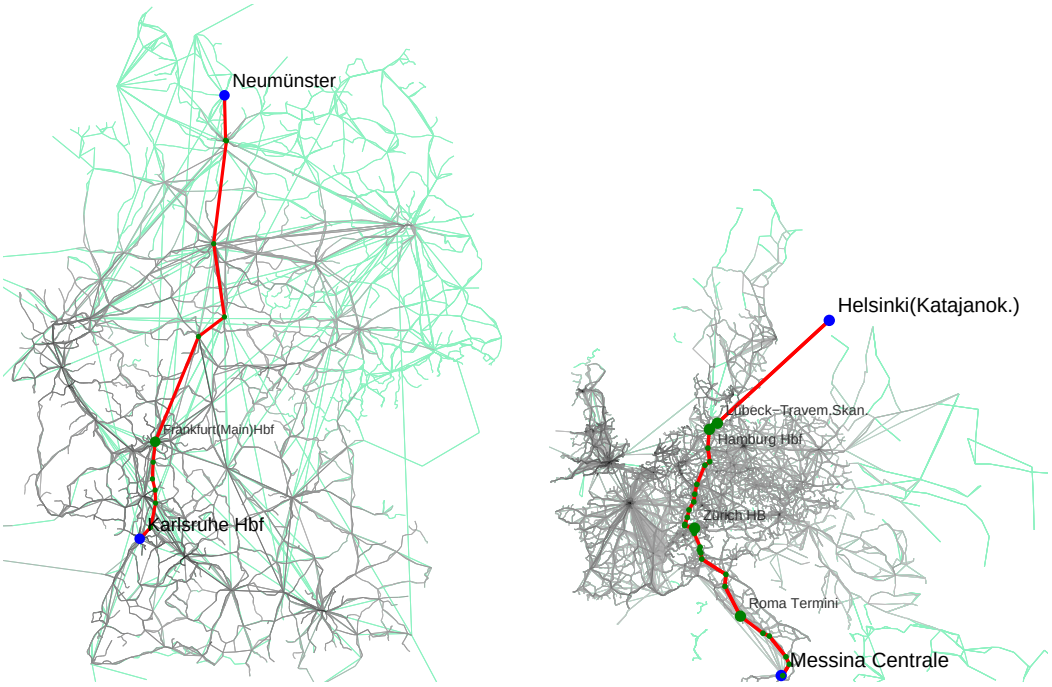
Figure 11: Which of the two nodes v and $w' = \text{pre}(w)$ should become the (new) predecessor of w?

w is smaller, decrease the key of w in in the priority queue, thus w is now reached through v instead from its prior predecessor $\text{pre}(w)$. If the distance of w would be the same whether w is reached through v or $\text{pre}(w)$, then $\text{pre}(w)$ is bent to v if and only if the function `decideSameKey` returns true for v and w. This turns out to play a crucial role in optimizing the quality of shortest paths in railway networks, since this situation turns up each time two shortest paths of the same length concur, and this is a very common situation in the expanded graphs. See Figure 11 for an illustration of the situation. We go into more detail about this topic in Section 4.4.

The nodes, which get touched by DIJKSTRA's algorithm are referred to as the *search space* of the algorithm. A node is called to get *settled*, if it has been dequeued from Q. A settled node v is never touched again during the execution of the algorithm (in the sense of getting reinserted into the queue), since all remaining nodes have a bigger distance from s than v. Thus, a shortest path has been found as soon as the target node t gets settled. The nodes which belong to the search space (and thus have been settled) are exactly all nodes $v \in V$ where $\text{dist}(v) \leq \text{dist}(t)$. If Q runs empty for some reason (without t getting settled), the target node is not in the search space, and hence no shortest path exists from s to t. Finally, the shortest path can be reconstructed by walking backwards along the path of preceding nodes, starting at the target node t until reaching the source node s. The complexity of DIJKSTRA's algorithm mainly depends on the data structure used for the priority queue. We use a very efficient implementation by Schultes [Sch07] which allows all operations to be executed in $O(\log n)$ time. This yields an overall complexity of $O\big((m+n)\log m\big)$ time, assuming $n = |V|$ and $m = |E|$.

Adaption

As mentioned in Section 4.2 the target node t is not known at the beginning of the shortest path query when stating realistic queries on the time expanded models. This leads to the only slight modification we have to apply to DIJKSTRA's algorithm in order to perform on our railway model. Since a node, which becomes settled is never touched again, and all nodes that are settled afterward have a greater or equal distance from the source node, we can abort the shortest path search as soon as a node v with $\text{station}(v) = T$, where T is the target station, has been settled. Thus in line 5 the abort condition has to be changed to $\text{station}(v) = T$. Of course the target station needs to be passed as an argument to the algorithm (instead of the



(a) Karlsruhe to Neumünster at 8:00 in the morning: 443 minutes travel time with 1 transfer

(b) Messina to Helsinki at 10:00 in the morning: 4651 minutes travel time with 4 transfers. The last part is done by ship.

Figure 12: Sample queries with the plain Dijkstra algorithm.

target node) for this to happen.

Figure 12 shows two example queries with the plain Dijkstra algorithm. The search space is visualized through the gray edges. The darker the gray tone, the more edges in the expanded graph between the two respective stations have been touched by the algorithm. Edges which are light green have not been touched at all. As we see, most of the edges in the condensed graph are gray, which means that at least one edge in the expanded graph has been touched by the algorithm. Please note again that all nodes (and thus edges between them) are touched which can be reached in time less than the target node. In our case this means, a pure Dijkstra search visits almost the whole graph, for example in the Europe graph the whole UK has been touched, though probably no connection ever would go through the UK if we want to go to Helsinki.

The itinerary obtained by evaluating the shortest path from Figure 12a is shown in Figure 13. It shows all the stops on route, as well as the trains of the elementary connections throughout the way.

In our very basic implementation of Dijkstra's algorithm we did not care about the `decideSameKey` operation, which chooses the preceding node if two shortest paths have equal length. In fact, it is just left blank, so the outcome which of the nodes gets the new predecessor is not cared about. Although, our primary optimization criterion remains travel time, we can

```
Karlsruhe Hbf        [10:44]  ->  Bruchsal             [10:55]     IR 02572
Bruchsal             [10:56]  ->  Heidelberg Hbf       [11:12]     IR 02572
Heidelberg Hbf       [11:14]  ->  Weinheim(Bergstr)    [11:27]     IR 02572
Weinheim(Bergstr)    [11:28]  ->  Bensheim             [11:36]     IR 02572
Bensheim             [11:37]  ->  Darmstadt Hbf        [11:47]     IR 02572
Darmstadt Hbf        [11:49]  ->  Frankfurt(Main)Hbf   [12:06]     IR 02572

Frankfurt(Main)Hbf   [12:57]  ->  Kassel-Wilhelmshöhe  [14:18]    ICE 00578
Kassel-Wilhelmshöhe  [14:20]  ->  Göttingen           [14:39]    ICE 00578
Göttingen            [14:41]  ->  Hannover Hbf         [15:15]    ICE 00578
Hannover Hbf         [15:18]  ->  Hamburg Hbf          [16:32]    ICE 00578
Hamburg Hbf          [16:42]  ->  Hamburg Dammtor      [16:44]    ICE 00578
Hamburg Dammtor      [16:47]  ->  Neumünster           [17:28]    ICE 00578
```

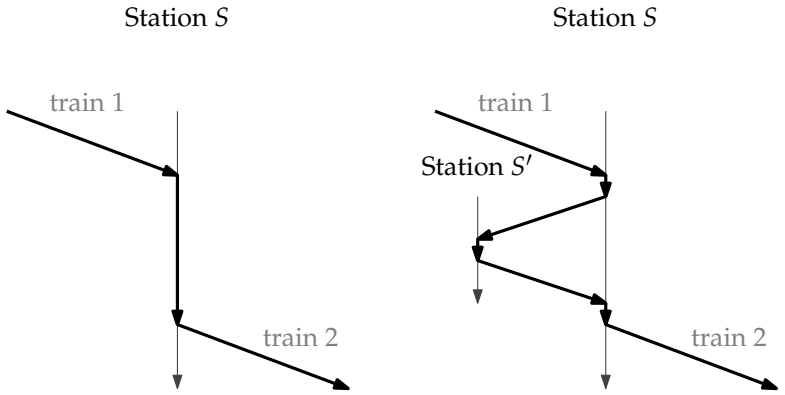
Figure 13: Itinerary calculated by DIJKSTRA's algorithm.

do some further optimizations, which are only in effect as long as they do not violate an optimal result regarding time. This can be done through carefully defining the behavior of the `decideSameKey` function. We go into more detail about this issue in the next section.

4.4 The Choice of decideSameKey

Have a look at Figure 13 again. The time between the arrival and the departure at Frankfurt (Main) is 51 minutes. Intuitively, we would immediately say that it makes sense to stay at Frankfurt and wait until the ICE train departs. In the expanded graph this equals taking a path from the arrival node to one of the transfer nodes, then walking along the consecutive transfer nodes until we reach the transfer node belonging to the departure event of that ICE 00578. From here on we turn off toward the departure node and thus board the desired train. However, the plain DIJKSTRA algorithm has no information about its current node being a transfer or some other node. It just selects its shortest path based on the distance from the source node. This may lead to very unpleasing results.

Figure 14 shows an example of what can happen if no care is taken. Obviously, both paths are shortest paths in the sense that they share the same travel time. However, in the figure to the left we do not wait at the station until train 2 departs, but instead take some other train to get to station S' and immediately get back to S, just in time to reach train 2. This is clearly an unwanted result. For that reason, we have to examine the node where the two shortest paths merge. This is one of the transfer nodes—let's call it v—where the paths from station S' joins back to S. Let u be the arrival node at S of the path that comes from S'. Assume the desired path to the left has been computed first, and now the node u gets settled in the priority queue. It sees that the node v (which is reachable from u) has already been touched (with its predecessor being another transfer node "above" v), and that $\text{dist}(v) = \text{dist}(u) + w(u, v)$. In this case `decideSameKey` gets called, and we want it to keep its current predecessor and not



(a) The desired outcome: Wait at S until train 2 departs.

(b) An equally valid shortest path: Hop over to some other station S'.

Figure 14: Two possible shortest paths in the expanded graph. Computing the desired path on the left is a matter of luck, if `decideSameKey` is not implemented carefully.

select u as its (new) predecessor. Two strategies how this can be accomplished are shown in the next two subsections, namely minimization of transfers and minimizations of geographical travel distance.

4.4.1 Transfer Minimization

The *transfer minimization* strategy prefers a path with less transfers over a path with more transfers. For that reason, a map $\mathsf{transfers} : V \to \mathbb{N}$ is defined, which maps each node v to its number of transfers it has on the (current) shortest path from s to v. A *transfer* is thereby defined as the number of edges $e = (u, v)$ along the shortest path from s to v for which $\mathsf{type}(u) = \texttt{Transfer Node}$ and $\mathsf{type}(v) = \texttt{Departure Node}$ holds. This can be imagined as a transfer corresponding to boarding a *new* train. For that reason, it is not possible to apply this strategy to the simple model, since in the simple model there is no way to distinguish between boarding a new train or staying in the same train.

The number of transfers can be computed in-place, so the overhead when performing the Dijkstra query is kept low. For ease of notation, let $\alpha : V \times V \to \{0, 1\}$ for two arbitrary nodes $u, v \in V$ indicate, whether there is a transfer when going from u to v. Each time a new node w is discovered from some node v, we set $\mathsf{transfers}(w) := \mathsf{transfers}(v) + \alpha(v, w)$. When w has already been touched and therefore gets rediscovered from v, `decideSameKey` tells whether v should be updated (predecessor changed to v) or not. If it does, we set $\mathsf{transfers}(w) := \mathsf{transfers}(v) + \alpha(v, w)$ again, otherwise leave $\mathsf{transfers}(w)$ untouched.

The implementation of the `decideSameKey` operation is very simple. It only needs to check whether $\mathsf{transfers}(v) + \alpha(v, w) < \mathsf{transfers}(w)$. If so, than it returns `true`—and therefore v becomes w's new predecessor—otherwise it returns `false` and nothing is changed.

4.4.2 Minimization of Travel Distance

A different approach of avoiding a situation as shown in Figure 14 is not minimizing transfers but minimizing the accumulated geographic length of the shortest path. The *geographic length* of a path $P = v_1, \ldots, v_k$ is defined as the sum of the geographic distances of each edge on the path. Thereby, the geographic distance of an edge connecting two nodes $e = (u, v)$ with $\text{station}(u) = \text{station}(v)$ is zero, otherwise it is the Euclidean distance between the two stations as defined in Section 3.3 and is denoted by $\text{dist}(\text{station}(u), \text{station}(v))$. So we define the geographic length of a path P formally by

$$\text{length}_{\text{geo}}(P) = \sum_{i=1}^{k-1} \begin{cases} \text{dist}(\text{station}(v_i), \text{station}(v_{i+1})) & \text{if } \text{station}(u) \neq \text{station}(v) \\ 0 & \text{otherwise} \end{cases}$$

Both paths from Figure 14 have the same distance regarding time, but the path to the right clearly has a bigger geographic length, since it has the geographic length of the left path plus two times the distance $\text{dist}(\text{Station 1}, \text{Station 2})$.

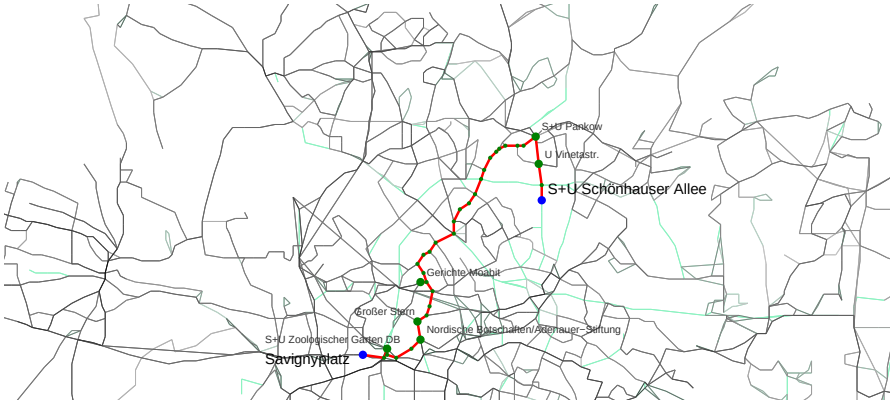
The implementation of this strategy is similar to transfer minimization. In addition to the value $\text{dist}(v)$, which holds the (time) distance from s for each node, we keep another value $\text{dist}_{\text{geo}}(v)$ which refers to the geographical distance of v from s. dist_{geo} can be computed in-place at the same time dist is computed or updated. The `decideSameKey` operation now prefers the node which has the smaller geographic distance, and thus for two shortest paths with equal length the one with smaller geographic length is selected. As opposed to transfer minimization, this strategy can also be applied to the simple model, since it does not require any of the additional features of the realistic model.

Figure 15 shows the difference that can be achieved through transfer minimization. It is, as can be seen in the figure, well possible that a whole different itinerary is computed for the same query.

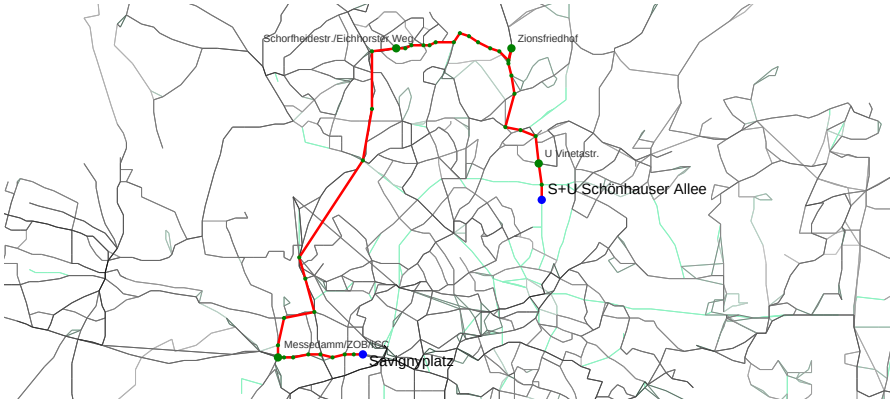
We have seen several strategies for implementing the `decideSameKey` operation. Unlike in road graphs, timetable networks demand much more attention to this issue due to the tremendous number of shortest path of equal length for each query. The two approaches we presented here are just examples of what can be done. We could, of course, think of many further strategies such as minimizing the number of stops (stations) or minimizing the number of "slow" trains to prefer express trains. However, since our presented strategies only decide which path to take, when two shortest path merge together at some node, there is no difference in the number of touched or settled nodes in the graph. Therefore, the search space (and thus the query time) is much the same, no matter which of our strategies we apply.[12]

In real world scenarios one usually has a centralized server that handles all the shortest path requests. Clients can then state requests to that server—for example via the internet. In such scenarios it is very important that these requests can be handled efficiently. For example, during the query from Messina to Helsinki in Figure 12b the (pure) DIJKSTRA algorithm has

[12] Of course we could think of a `decideSameKey` operation that is very intensive, thus yielding worse query times or even touching nodes which normally would not be touched by the algorithm.



(a) No strategy applied



(b) Transfer minimization

Figure 15: A query from Savignyplatz to the subway station Schönhauser Allee in the bus network of Berlin. Each journey takes the same time, but the second one has two transfers less leading to a completely new route.

touched 4.779.492 nodes and 6.829.885 edges in the expanded graph taking almost 3 seconds. Moreover, almost the whole graph is drawn gray, which means the algorithm advanced into regions one would never expect an optimal itinerary to go through. One can almost certainly rule out that an optimal itinerary would go through London for example. This is very unsatisfying. To overcome this, speed-up techniques have been developed to reduce the search space which we go into more details with in the next section.

5 Speed-Up Techniques

In this section we study some speed-up techniques, which mostly have been optimized to work well with road networks. We go into more details with Arc-Flags, the ALT algorithm, and a combination of these two techniques. In the section afterward we present results how these

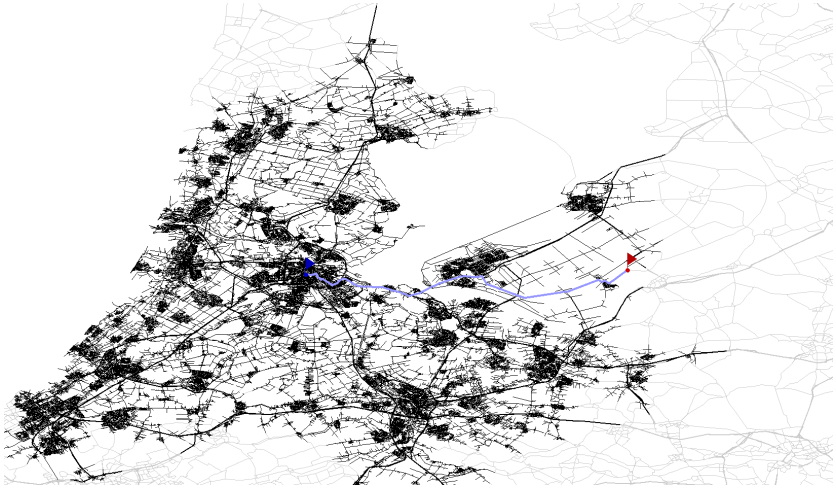
techniques behave, and observe that in timetable networks the speed-up techniques behave quite differently to road networks. For example Arc-Flags gives much worse results than in road networks. However, the combination of Arc-Flags with ALT plays off very well, in contrast to road networks where combining these techniques gives almost no advantage at all [Sch08].

The theoretical complexity of DIJKSTRA's algorithm is $O((m+n)\log m)$ which one would think of pretty good at first glance. Of course this is true because it allows us handling huge sets of input data at all. So why care about speeding up DIJKSTRA? The roots of the research for making DIJKSTRA's algorithm faster indeed come from public transport networks and were first introduced by Schulz, Wagner and Weihe in 1999 [SWW99]. The development then, however, started to focus more on road networks which is mainly due to the availability of large road network graphs to the public. At the 9'th DIMACS challenge in 2006 [DGJ06] which had speed-up techniques as its topic, most submissions were therefore dedicated to road networks. Furthermore, speeding up DIJKSTRA is easier on road networks, hence most attention has been payed to this matter in the first place.

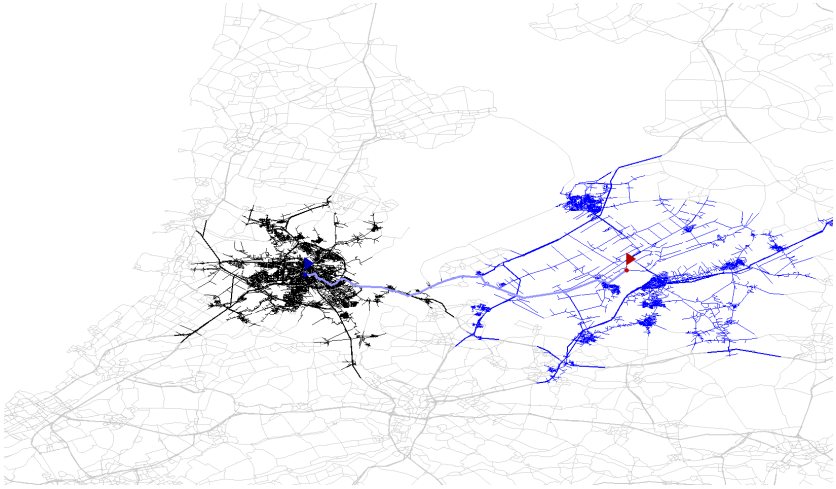
Several techniques have emerged, which yield really amazing results on road networks, from those the fastest ones known today being transit node routing [BFSS07] and SHARC [BD08]. A good overview about the most important speed-up techniques is given in [WW07] by Wagner and Willhalm and in [SS07] by Sanders and Schultes, which can be seen as a good introduction into the matter. We do not go into much detail about each and every speed-up technique but focus on two main techniques, which can be adapted to railway networks relatively easy.

The first one we study is called Arc-Flags and has been proposed by Lauther in [Lau04]. It has been studied and improved further by several authors in [MSS+05, KMS05, HKMS06, Hil07]. The basic approach is as follows. Informally, it basically partitions the graph into regions, and to every edge (or arc) a bit field is assigned with its i'th entry being `true` if and only if a shortest path toward the i'th regions uses the respective edge. During queries, edges having their flag set to `false` regarding the target region can be ignored. In a preprocessing step the partition and the arc flags are computed, which are then used during the actual queries. There are further speed-up techniques which are based on graph partitions [MSS+06]. However, Arc-Flags has proven to be the best one in road networks, so we focus on Arc-Flags in this work.

The second speed-up technique we go into more detail with, is the ALT algorithm. It is a variation of the A^* algorithm [HNR68] which has been introduced by Goldberg and Harrelson in [GH04]. Some further research has been put into this approach by Delling and Wagner in [DW07]. Basically, it is a goal directed search, exploiting the triangular inequality according to some carefully chosen landmark nodes in the graph. As a consequence, nodes which lie between the source and the target node are prioritized in the priority queue. This technique also requires some preprocessing for computing the landmark nodes, but since we do preprocessing on the much smaller condensed graph here (in contrast to Arc-Flags, where we compute the flags on the expanded graph), preprocessing time is almost negligible.



(a) Unidirectional Dijkstra



(b) Bidirectional Dijkstra

Figure 16: Search space of a query around Amsterdam in the Netherlands' road network. The upper picture shows a plain Dijkstra search, while the lower picture shows a bidirectional search.

Before we go into more detail with these two speed-up techniques in the next two sections, we want to make a few notes about the most obvious technique for speeding up queries: Bidirectional search. The basic version of Dijkstra's algorithm does a *forward search* alone. This means, for some s-t query in a graph $G = (V, E)$ we start at the source node s and the algorithm terminates as soon as the target node t has been settled. Now let $\bar{G} = (V, \bar{E})$ be the backward graph obtained from G when all edges are flipped. Instead of initiating only one search from s to t in G we start another search in the backward graph $\bar{G}$ from t to s at the same time, called the *backward search*. As soon as the search spaces meet, meaning there is a node v which has been settled in the forward and the backward search, the algorithm may

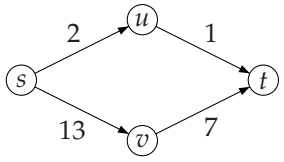


Figure 17: Simple example to illustrate "unnecessary" edges. The edge (s, v) is on no shortest path from s to t and thus can be ignored during the DIJKSTRA search.

be aborted, and the shortest path is the concatenation composed of the two paths from/to v. Figure 16 shows the difference in the size of the search space on a road network query when a bidirectional search is used over a simple unidirectional search.

Unfortunately, as we worked out in Section 4.2 the exact target node is not known for the earliest arrival problem, since we do not know at which time we arrive at the target station. Thus, we have no basis from where we should start the backward search, rendering the bidirectional approach impossible. As a consequence, any speed-up technique which relies on a backward search, especially hierarchical approaches, can not be used with our railway model. Even worse, some unidirectional techniques like the basic implementation of the ALT algorithm also need the target node as input. However, in this case the algorithm can be modified to work in our scenario where we do not know the exact target node.

5.1 Unidirectional Arc-Flags

When we try to engineer DIJKSTRA's algorithm, one central insight is that the algorithm does not need to touch each and every node and edge of the graph in order to find a shortest path toward the target node t. This allows some room for improvement by trying to reduce the number of touched nodes. When we look at Figure 17, we can see that the edge from s to v is certainly not used by a shortest path beginning at s with target node t. Thus, this edge does not need to be considered for a shortest path query from s to t. More generally, the edge can be ignored for *every* shortest path with target node t at the point where DIJKSTRA's algorithm is processing the node s (We could imagine the graph being larger, and during the computation of the shortest path at some point arriving at s).

What we do now, is that in a *preprocessing step* we compute "unnecessary" edges which can be ignored during the shortest path search, eventually yielding a smaller search space. Let $e = (u, v)$ be an arbitrary edge in E and V_e be the set of all nodes that can be reached through a shortest path starting with u over e. Then, when computing shortest paths from s to t, all edges e where $e \notin V_e$ can be ignored. In our example the set $V_{(s,v)}$ only consists of the node v, since for every other node in the graph a shortest path from s does not use the edge (s, v), but the edge (s, u) instead.

Storing all sets V_e for a graph G would lead to space complexity $O(nm)$ which is much too large for our graphs to be held in memory. Hence, we partition the graph into r different regions with $r \ll n$. Let region : $V \to \{1, \ldots, r\}$ be the function that maps each node to its region-id. Then, instead of assigning each edge e its set V_e of nodes that can be reached

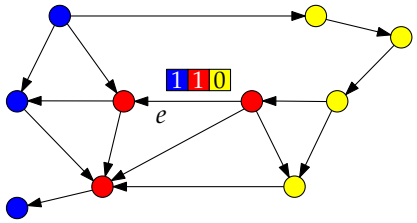


Figure 18: Sample edge with arc flags vector. Regions are marked with different colors. Since the yellow region can not be reached by a shortest path starting from e its flag is set to `false`.

through a shortest path starting at e, we assign a subset $R_e \subseteq \{1, \dots r\}$ of region-ids of exactly those regions which can be reached through a shortest path starting at e. We say, that a region with id r can be reached (through a shortest path beginning with e), if at least one node $v \in V_e$ with $\text{region}(v) = r$ exists.

The implementation is done using a bit-vector of size r for every edge e. The i'th entry of the vector is set to `true` if and only if at least one node in region i can be reached through a shortest path beginning at e (And `false` otherwise). See Figure 18 for an illustration. The DIJKSTRA search from s to t can now be reduced to the subgraph induced by those edges which have the entry of $\text{region}(t)$ set to `true` in their bit-vector. A proof of correctness can be found in [HKMS06]. The additional space required to store the arc flags is in $O(rm)$, for sparse graphs—like our expanded graphs—this even leads to space complexity $O(rn)$. The number of regions r gives us a trade off between speed-up and space consumption. The higher r is chosen, the more arcs can be ignored during the DIJKSTRA query. The optimal case is setting $r = n$, thus only edges which are part of shortest paths toward t would be considered during a search from s to t. The other extreme, when making r very small, leads to a much higher probability that the relevant flags of many "unnecessary" edges are set to `true`, even though they are not contained in a shortest path to t, since chances are high, that at least one other node in the target region exists from which a shortest path is using the considered edge.

The edge induced subgraph due to the arc flags of the particular target region can be computed on the fly when executing the algorithm for each query. In line 7 of algorithm 1 we just need to check whether the flag regarding our target region is set to `true`. If it is not, the edge can be simply ignored and its tail node does not need to be examined for insertion into the priority queue.

The Preprocessing

For having the arc flags available when executing shortest path queries, we need to compute them in a preprocessing step. This step only needs to be done once, since the computed arc flags can be used for all subsequent queries as long as the graph structure does not change.[13] The preprocessing consists of the following two steps.

1. Partitioning and

[13] This makes this technique somewhat unsuitable for dynamic scenarios, where weights or even the graph's structure may change often—but this is not a topic in this thesis.

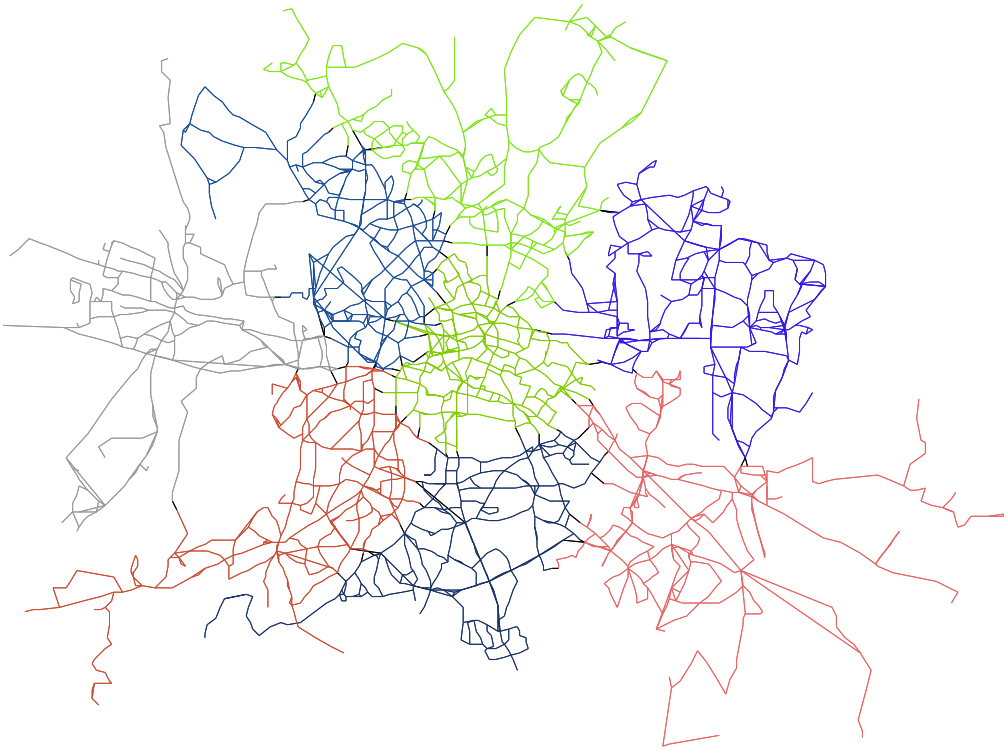


Figure 19: Berlin's Bus Network partitioned into eight regions by SCOTCH.

2. computing the arc flags due to the regions from the previous step.

The partitioning is done on the condensed graph, because during a query we do not know the exact target node in the expanded graph. Thus, we also do not know the target region which would render the whole Arc-Flags approach useless. But, since we know the target station, we simply assure that all nodes belonging to the same station also belong to the same region. This can be achieved through partitioning on the condensed graph. So in a first step we retrieve a map region : $\mathcal{B} \rightarrow \{1, \ldots r\}$ which assigns each station its region-id. In a second step we expand the map to the nodes of the expanded graph such that each node of the expanded graph is mapped to the region-id of its respective station it belongs to; formally: $\text{region}(v) := \text{region}(\text{station}(v))$ for some node $v \in V$ from the expanded graph.

The choice of the partition type has a big influence on the efficiency of the speed-up technique. One goal is to balance the regions well, meaning they all contain a similar number of nodes. Furthermore, the number of edges that cross region borders should be as low as possible. In [HKMS06] and [MSS$^+$06] several partitioning methods are presented, some of which need geographic information or some other embedding of the graph in the plane.[14] For our experiments we solely use SCOTCH [Pel07] which provides a family of (multilevel) partitioning algorithms developed at the Laboratoire Bordelais de Recherche en Informatique

[14] Our (condensed) graphs are equipped with geographic information.

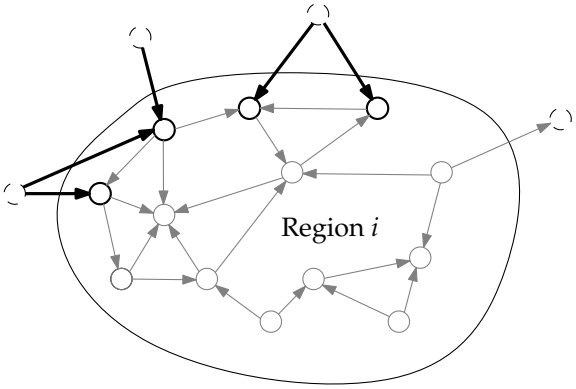


Figure 20: A region i with some boundary nodes (the fat ones). The fat edges are the overlapping arcs through which the region can be entered.

in France. Figure 19 shows the result of a partition with eight regions on the graph of the local bus network of Berlin.

The second step is to compute for each edge the actual arc flags based on the partition layout. A naive way of doing this is computing a one-to-all shortest path tree from every edge $e = (u, v)$. This can be done by a standard DIJKSTRA algorithm which does not stop until all nodes have been settled [Lau04]. The start node is set to u and every time a node w gets settled during the algorithm, we check whether e is on the shortest path from u to w. Accordingly, the arc flag $f_e(\text{region}(w))$ is set to true if and only if e is contained on the path from u to w. The information whether e is contained on the path can be propagated through all nodes during execution of the algorithm as follows. We assign a flag to each node which indicates whether e is used on the shortest path to the specific node. Each time we set the predecessor of some node w during the algorithm, we set the e-flag to the value of its preceding node, unless we used e directly to get from $\text{pre}(w)$ to w. In this case the e-flag must be set to `true`. This way, and assuming that we initialize the e-flag of the source node u with `false`, we can easily determine whether the arc e is contained on the shortest path from u to w with almost no overhead. However, computing m shortest path trees takes $O(m((m+n)\log n))$ time. Assuming the graphs are sparse[15]—and therefore $m \in O(n)$ holds—we still get $O(n^2 \log n)$ time complexity which is far too much to be practical. Preprocessing on huge graphs such as the German or even the European rail network would take weeks on todays high end computers. For that reason, we use a smarter technique for computing the arc flags which does not require an all pair shortest paths computation.

In the previous approach we did not really exploit the partitioning scheme for our preprocessing. We simply computed shortest paths from each arc $e \in E$ to *every* node in the graph. But we can do better. Let i be the region-id of the i'the region from $\{1, \ldots, r\}$. Then every (shortest) path from any node s (which is not a member of the region i) to any node inside the region i has to *enter* the region at some point. Thus, let's call an edge $e = (u, v)$ with $\text{region}(u) \neq i$

[15] In our scenarios they are.

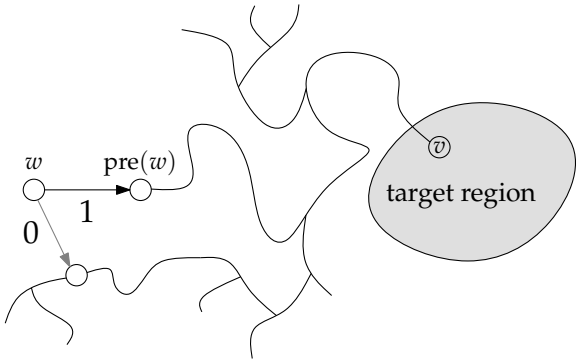


Figure 21: Illustration of the construction of the backward-DIJKSTRA shortest path tree from some boundary node v. The gray edge is not contained in the shortest path tree, and therefore its flag is set to `false`.

and $\text{region}(v) = i$ an *overlapping arc*. The set $B(i) := \{v \mid \text{There is an overlapping arc } e = (u, v) \text{ into } i\}$ is called *set of boundary nodes* of the region i, and its members are called *boundary nodes*. In [MSS+06] Möhring, Schilling, Schütz, Wagner and Willhalm have shown that it is sufficient to only compute shortest paths to the boundary nodes of each region. See Figure 20 for an illustration of the boundery nodes of a region.

For that reason, we no longer use a forward-DIJKSTRA on G to compute the shortest path trees, but instead for each region i and every boundary node $v \in B(i)$ we compute a one-to-all shortest path tree on the backward graph $\bar{G}$ of G starting at v. Now every time a node w in $\bar{G}$ is settled by the backward-DIJKSTRA, we acquire its preceding node $\text{pre}(w)$ in $\bar{G}$ and the flag of the corresponding forward-edge $(w, \text{pre}(w))$ in G is set to `true` due to the region i. Please refer to Figure 21 for an illustration. This method gives us a significant speed-up in preprocessing time. The amount of time reduced heavily depends on the quality of the partition. Since our partition is computed on the condensed graph and is then mapped over to the expanded graph, region borders always go along the "border-nodes" of the stations. These mostly consist of arrival and departure nodes which have train connections going outside the region. One can easily see, that this is not necessarily the best way to partition the expanded graph in regard to a minimal number of boundary nodes. Therefore, the number of the overall boundary nodes is still very large in our scenario. Preprocessing times are shown in Section 6, but they usually vary between several hours up to more than a day depending on the graph size and number of regions.

Importance of `decideSameKey`

As we have worked out in Section 4.4 the number of different shortest paths between two nodes in the expanded graph is tremendous. Of course these considerations also apply to the preprocessing step when computing the one-to-all shortest path trees in $\bar{G}$. As a consequence, if we do not choose the strategy for the `decideSameKey` operation carefully in the preprocessing-DIJKSTRA, we end up having almost all arc flags set to `true`, and thus not gaining anything when stating queries with these flags.

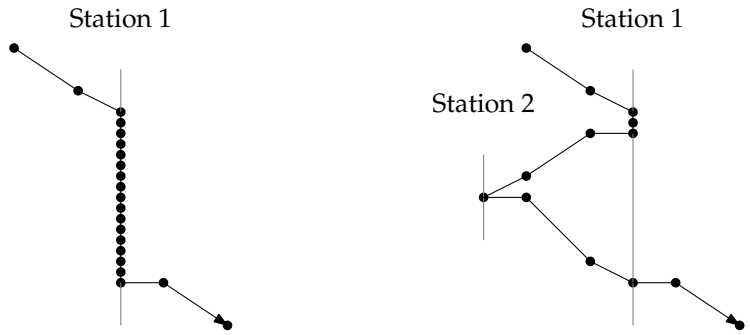


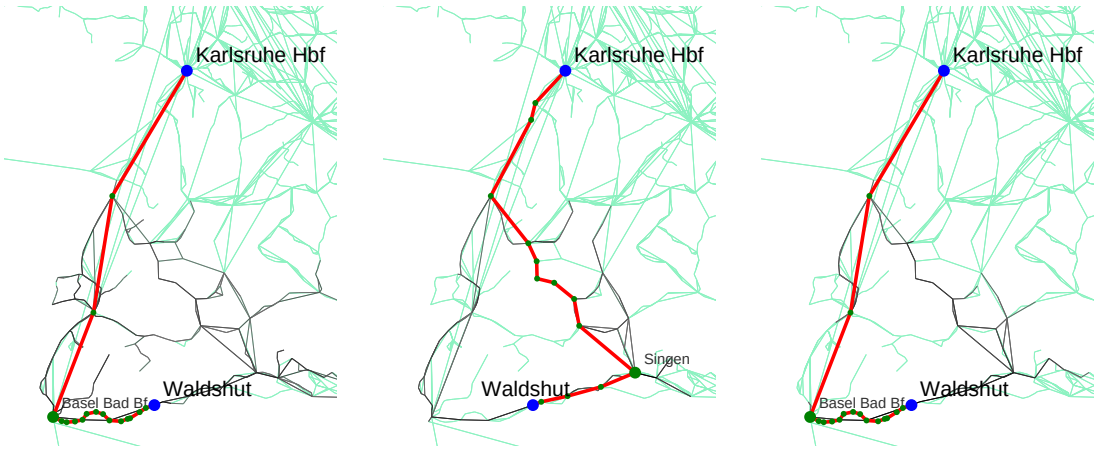
Figure 22: The left path has a lot more hops than the right path. However, the left path should be preferred.

When we do shortest path computations on road networks and get the situation of two shortest paths merging together[16], and thus `decideSameKey` gets called, it usually makes sense to choose the shortest path with less nodes (hops) on it. This has the semantics of preferring a route with less crossings over a route with more crossings, even though they may be equally fast. This strategy does not lead to good results in the time expanded graph of our model. Imagine arriving at some station S_1 with lots of traffic (for example in some big city). Probably we switch trains here, and thus have to go through a *lot* of transfer nodes in order to get to the desired train. Figure 22 shows that in this case it is more attractive for the algorithm to use a path which leaves S_1 for S_2 and gets back immediately. This is due to the enormous number of transfer nodes we have to go through at S_1.

At first glance, it seems to be intelligent to use the same strategy for `decideSameKey` as we use later for our queries. However, this has proven to result in very bad speed-ups. When using transfer minimization we yield almost no speed-up. The number of settled nodes improves by a few hundred where the scale of settled nodes of our queries is around 100,000. This means, that during the preprocessing almost *all* arc flags have been set to `true`. Also, the strategy which minimizes travel distance does not lead to good results (though they are a little bit better than transfer minimization). Please see Section 6 for a comparison of the different strategies.

For achieving any significant speed-up at all, we have to do some really aggressive optimization. By aggressive we mean, that we need to force as many flags to be set to `false` as possible. That is, if a flag of some edge is kept `false` we must try to prohibit as strongly as possible that it gets overwritten to `true` by some other backwards-Dijkstra-run from the same region. Thus, the criterion which the decision at the `decideSameKey` operation is based on, should have the same outcome as often as possible, no matter from which boundary node of the respective region we started our Dijkstra search. To achieve this, we use the *direct geographical distance* between the the source boundary node s and the nodes involved in the `decideSameKey` operation as the determining factor. If for some node v (which has $\text{pre}(v)$ as its current predecessor) that is reached from some other node u we have to decide which

[16] Which in fact does not really occur very often in road networks, so here it is even sufficient to do nothing about this matter at all.



(a) Without Arc-Flags acceleration **(b)** Alternative 1: Departing at 11:19 **(c)** Alternative 2: Departing at 15:15

Figure 23: Comparison between Arc-Flags using "geographical distance to target" strategy for preprocessing and two alternative shortest path routes between Waldshut and Karlsruhe. While plain DIJKSTRA visits a lot of unnecessary arcs, with Arc-Flags only those edges are visited which go along routes for which, at some point in time, no better (in the sense of being more directed toward the target region) route exists.

one becomes the "new" predecessor, we choose the node due to the following criterion.

$$\text{pre}(v)_{\text{new}} := \underset{w \in \{u, \text{pre}(v)\}}{\text{argmin}} \{\sqrt{\left(\text{coord}_x(w) - \text{coord}_x(s)\right)^2 + \left(\text{coord}_y(w) - \text{coord}_y(s)\right)^2}\},$$

where s is the source boundary node where the backward-search started from. This can be imagined as follows. At each node we always try to geographically go further toward the target region, thus cutting off routes which do not lead toward the target region. However, if (for example at some different time of day) valid connections are *only* available trough some other route, the arc flags are opened (set to `true`) for both alternatives. Therefore, when stating a query, both alternatives are explored by the DIJKSTRA algorithm, regardless of the time at the source station we stated our query at. This can be seen nicely in Figure 23 which shows a cut-out of the German railroad network. There are two routes from Waldshut to Karlsruhe. Which one is faster depends on the time we want to start our journey in Waldshut. Thus, only these two routes are explored, but other routes, where no shortest paths lead over, are not touched.

Figure 24 shows a general comparison between the three different `decideSameKey` strategies. Each query starts at 8:00 in the morning and the queries with arc flags are using the same 128 region partition generated by SCOTCH. During the queries we applied the strategy of transfer minimization according to Section 4.4.1. We can see that depending on the strategy chosen for the `decideSameKey` operation in the preprocessing phase, there are *huge* differences regarding the number of settled nodes and touched edges in the shortest path queries after-

ward. Whereas "distance" and "transfer" have almost no effect compared to a plain Dijkstra query, "geographical distance to target" gives a nice speed-up.

Disadvantages of Arc-Flags

Although we can achieve good speed-ups with the geometric strategy when using Arc-Flags, this also leads to some disadvantages. First, the preprocessing time is still way too long. On the graph of whole Europe with a large partition, preprocessing takes more than four days on our machines. Please refer to Section 6 for the measurements and the hardware specification we used in our experiments. This makes Arc-Flags very inflexible for dynamic scenarios where weights may change, or the graph needs to be patched up in some way (for example due to delays in the train network). Although, recently some promising development has come up to dramatically reduce preprocessing time. For example in [BD08] Bauer and Delling propose to contract the graph during the preprocessing phase, thus making the graph smaller, which eventually leads to shorter preprocessing times (and even more amazing speed-ups regarding query times). Another improvement is to use centralized shortest paths computations which reduces the number of shortest path that need to be computed and therefore also reduces preprocessing time, see [Hil07].

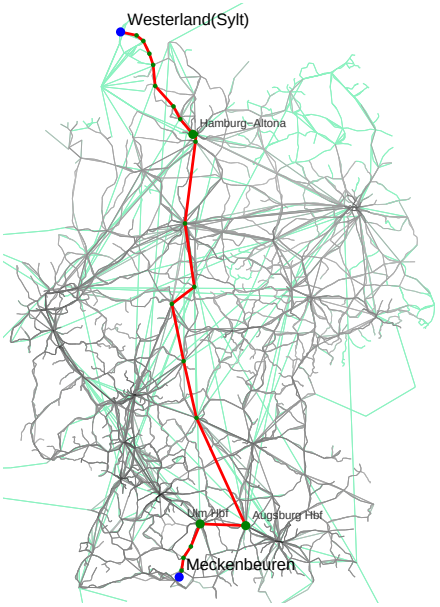
Concerning the second disadvantage, and this might weigh more heavily, the aggressive geographic optimization in the preprocessing destroys the room for second order optimization during the execution of shortest path queries as we described in Section 4.4. Even worse, optimizing geographically toward the target station (or region) can contradict the desire to minimize transfers or travel distance, because shortest paths with less transfers may be cut off with a `false` indicating arc flag. Thus, the optimal shortest path—in sense of the number of transfers—is lost.[17] This can lead to pretty bad itineraries as can be seen in Figure 24d compared to Figure 24a. While plain Dijkstra gives us a pretty nice itinerary with just three transfer, the Arc-Flags method gives us plenty more transfers, although we used transfer minimization in this query as well.

With Arc-Flags and the best strategy for `decideSameKey` applied, we optimize query times pretty well. However, the dimension in which optimizations are applied to, is only *geographically*. As we see in Figure 23 almost no edges which are not on train routes (containing optimal connections for some time of day) are considered, but on these branches of the network we still examine all connections in the time span between the start time at the source and the arrival time at the target station. Hence, the "time component" of our network is not optimized too well. The next speed-up technique we present, namely the ALT algorithm, has its focus on the time component. Instead of cutting off paths that go into the wrong direction geographically, we cut off paths that probably do not reach the target station in time.

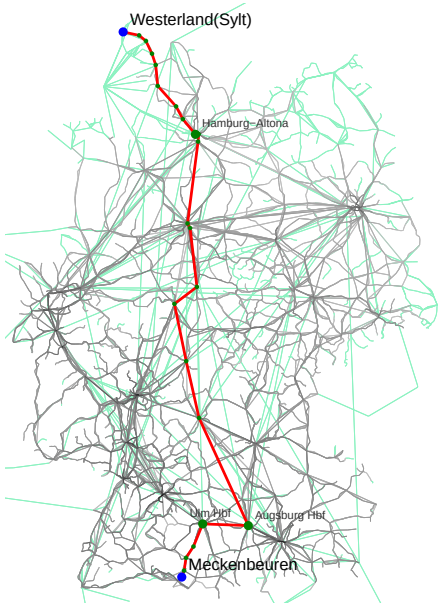
5.2 Unidirectional ALT

Another algorithm for solving the shortest path problem is the A* algorithm [HNR68]. However, we can easily reduce A* to Dijkstra's algorithm, and thus consider it as a speed-up tech-

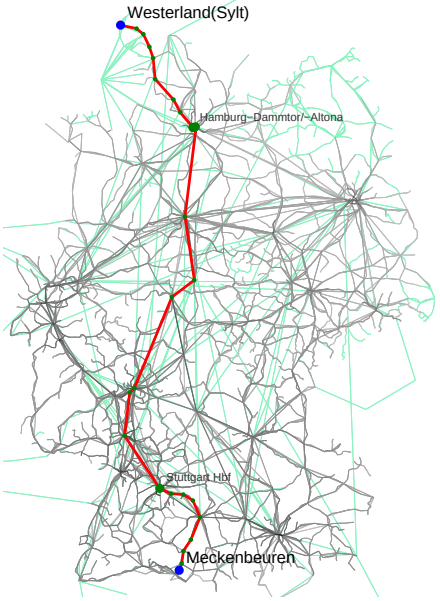
[17] Of course the now obtained shortest path (with possibly more transfers) is still optimal regarding travel time.



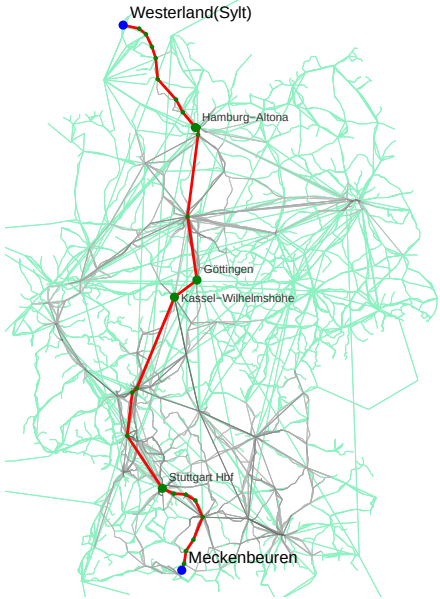
(a) Plain Dijkstra. Settled nodes: 432,770, touched edges: 623,234



(b) Arc-Flags with travel distance strategy. Settled nodes: 407,940, touched edges: 584,226



(c) Arc-Flags with transfer strategy. Settled nodes: 430,279, touched edges: 618,840



(d) Arc-Flags with geographic distance strategy. Settled nodes: 73,256, touched edges: 91,223

Figure 24: A query from Meckenbeuren to Westerland on Sylt.

nique of DIJKSTRA. We now derive some general ideas behind A*, referring to the terms of Goldberg in [GH04], which uses lower bounds to the target node to estimate the direction in which the shortest path search should advance. After that, we introduce landmarks which are basically special nodes in the graph which help us to compute good lower bounds. This leads us to the ALT algorithm, which is an abbreviation for "A*, Landmarks and Triangle inequality".

Potential Functions and A*

Consider $\pi : V \to \mathbb{R}$ to be a *potential function* from the nodes into the reals. When we modify the weight function w of our graph G to a new weight function w_π with $w_\pi(u,v) := w(u,v) + \pi(v) - \pi(u)$, then the length of every (not necessarily shortest) path between two arbitrary nodes s and t changes by the same amount $w_\pi(t) - w_\pi(s)$. This is because the potentials of subsequent nodes along the paths cancel out each other. As a consequence, finding a shortest path due to w is equivalent to finding a shortest path due to w_π. Since we cannot handle negative edge weights we call a potential function π *feasible* if $w_\pi(e) \geq 0$ holds for all edges of our graph. This implies the following two statements which can easily be verified.

(a) Given a feasible potential function π, and for some (target) vertex t we have $\pi(t) \leq 0$, then for any node $v \in V$ the equation $\pi(v) \leq \mathrm{dist}(v,t)$ holds.

(b) If π_1 and π_2 are two feasible potential functions, then $\max(\pi_1, \pi_2)$ is also feasible.

This can be seen as $\pi(v)$ being a *lower bound* on the distance between v and some fixed target node t in the graph. Furthermore, the second statement can be interpreted like this. If we always (meaning for each node) use the best lower bound we can get, we still have a feasible potential function which can be applied to DIJKSTRA's algorithm. Thus, from now on all of our potential functions are feasible, and we omit to mention the word feasible each time.

The modification of DIJKSTRA that needs to be applied in order to derive the A* algorithm is very simple. Instead of inserting new nodes v with key $\mathrm{dist}(v)$ into the queue, we insert them with respect to our potential function, and thus the key is altered to $\mathrm{dist}(v) + \pi(v)$. In this case, the search space is reduced to the nodes $v \in V$ where $\mathrm{dist}(v) \leq \mathrm{dist}(t) - \pi(v)$. This implies that the better the lower bound, meaning the greater the value of π, the smaller the search space grows. In the most simple case, which is the idea behind the original A* algorithm, the potential function π is defined as $\pi(v) := \mathrm{dist}_{\mathrm{geo}}(v,t)$—the straight geographic distance between v and the target node t.[18] Then in the priority queue nodes are prioritized (and thus get settled earlier) which have a smaller geographical distance to the target node than nodes with a higher distance. Hence, the search is goal directed, in the sense that it preferably scans paths leading toward the target node first. If we define $\pi(v) := \mathrm{dist}(v,t)$, the potential function yielding exactly the shortest path distance to the target node t, we would get the best possible lower bound, and thus only nodes along shortest paths would be settled during the DIJKSTRA algorithm. Of course, having such a potential function available would require a distance table of all node pairs held in memory somewhere, which is not practical at all.

[18] Note, that π is not a fixed function like the edge weights w, but depends (in this case because of the location of the target node) on the actual shortest path query that is being stated.

Furthermore, preprocessing time for computing all-pair shortest paths is way off the limits. So we have to find some other good feasible potential function.

Landmarks and the Triangle Inequality

The basic idea behind using landmarks is having a fixed number of acclaimed landmark nodes to/from which we precompute one-to-all shortest-path trees. Therefore, we have the exact distances from each node v to each of the landmarks and vice versa. Now, for a target node t, some node v and a landmark ℓ the following two equations hold by the triangle inequality.

$$\text{dist}(v,t) + \text{dist}(\ell,t) \geq \text{dist}(v,\ell) \tag{4}$$
$$\text{dist}(\ell,v) + \text{dist}(v,t) \geq \text{dist}(\ell,t) \tag{5}$$

Please also refer to Figure 25 for a visualization of this matter. To get a feasible potential from this coherence, we resolve the equations (4) and (5) to $\text{dist}(v,t)$ and thus get

$$\pi(v) = \max\{\text{dist}(v,\ell) - \text{dist}(t,\ell), \text{dist}(\ell,t) - \text{dist}(\ell,v)\} \leq \text{dist}(v,t)$$

which is a lower bound of the distance from v to t. To get the best possible lower bound we can consider all landmarks $L \subseteq V$ and use the maximum value from each triangle inequality, yielding

$$\pi(v) = \max_{\ell \in L} \max\{\text{dist}(v,\ell) - \text{dist}(t,\ell), \text{dist}(\ell,t) - \text{dist}(\ell,v)\}.$$

Although one might think that for the sake of the lower bound the more landmarks are taken into account for computation the better, but this has proven to be false. The overhead of computing the lower bounds outweighs the actual increase in speed of heaving a better lower bound in this case. Thus, we only use a fixed number of concurrent landmarks. These landmarks are determined before the execution of the actual DIJKSTRA search, and are selected due to providing the best lower bounds for the start node. We refer to these landmarks as *active*. Furthermore, the set of active landmarks is updated during the execution of the algorithm within some fixed interval, e.g. after a fixed number of nodes have been settled. This ensures that we maintain a reasonably good lower bound during the whole execution of the algorithm without the need of calculating the distances to all landmarks all the time.

Preprocessing

Again, we have the problem of not knowing the exact target node when stating a shortest path query. Thus, we can not use landmarks in the expanded graph as long as t is unknown. For that reason, we do preprocessing on the much smaller condensed graph, which is okay, since we defined the edge weights in G^* to be the lower bound regarding travel time between the two connected stations (nodes). When doing the DIJKSTRA search in the expanded graph, we then compute the potential function through

$$\pi(v) = \max_{\ell \in L} \max\{\text{dist}(\text{station}(v),\ell) - \text{dist}(T,\ell), \text{dist}(\ell,T) - \text{dist}(\ell,\text{station}(v))\},$$

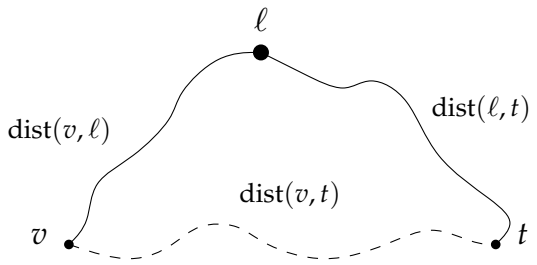


Figure 25: Visualization of the relation between the nodes v, t and ℓ and (shortest) paths between them. The triangle inequality can be applied in multiple ways according to the equations (4) and (5).

where T is the target station of our query. Of course, since in this case the distances are, again, only lower bounds (because they are computed on the condensed graph), the potential function is not as good as if we would use exact distances from the time expanded graph.[19]

The selection of the landmark nodes is crucial to the performance of the ALT algorithm. The most naive approach, using random nodes as landmarks already gives pretty good results, but we can do better. The triangle inequality yields better lower bounds the "flatter" the triangles are. Thus, the best results can be achieved if the landmark nodes are right "behind" or "in front of" the target node t. One way to increase the chance of having this result, is to place landmark nodes along the outer boundaries of the graph. A few techniques that try to achieve this effect are presented in [DSSW06], from which we use MaxCover.

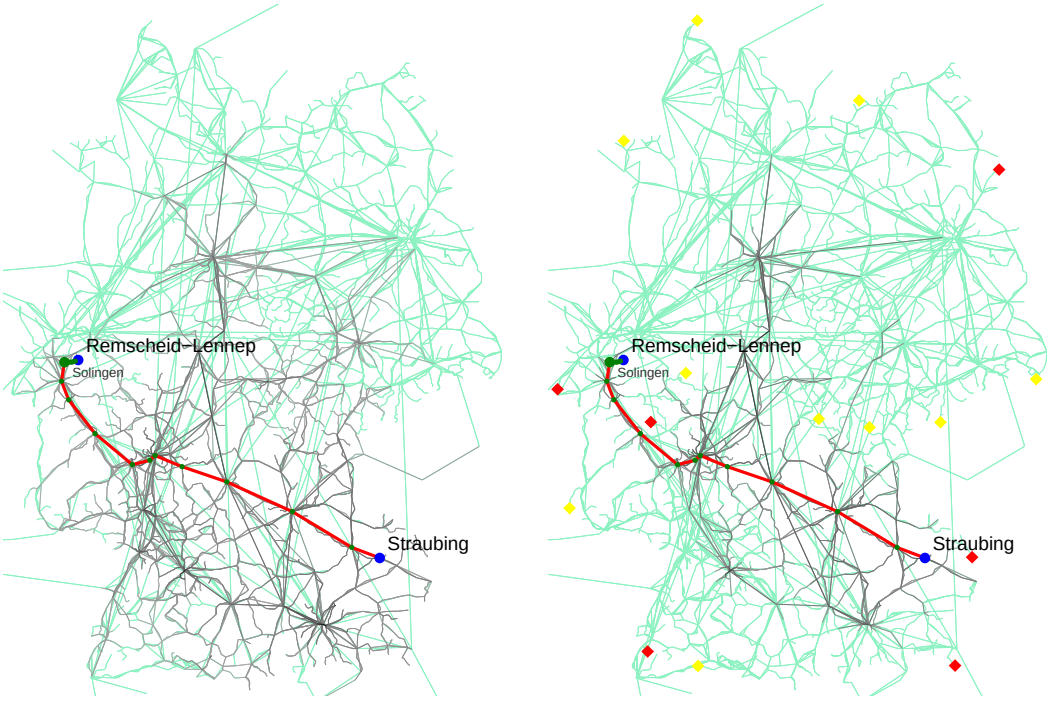
The time complexity for preprocessing the ALT algorithm is *much* less than for Arc-Flags. This is because we only use the condensed graph to compute the landmarks, which is much smaller than the expanded graph. For example, the condensed graph of Germany has only 6,730 nodes, whereas the expanded graph has 1,661,828 nodes. This decreases the time for computing a one-to-all shortest path tree dramatically. Hence, preprocessing time becomes negligible for the ALT algorithm, taking only a few seconds on the time expanded railway model.

Discussion

Figure 26 shows an example query in the German railroad graph when using the ALT algorithm with 16 landmarks compared to the same query using plain Dijkstra. As we see, the search space is considerably smaller than with plain Dijkstra, but the "visual difference" in the figure does not seem to be as dramatic as with Arc-Flags (and our best `decideSameKey` strategy). This can be put down to the property of the ALT algorithm preferring nodes with a shorter distance to the target node instead of the wrong geographic direction as Arc-Flags does. Hence the number of touched connections per edge in the visualized graph is optimized but there might still be some touched edges which point to the wrong geographical direction.

For this matter have a look at Figure 27. When stating a plain Dijkstra query we visit *all* nodes $v \in V$ with $\text{dist}(v) \leq \text{dist}(t)$. This has the effect that connections to stations are touched even if we already have reached the specific station by an earlier connection. In our

[19] We can think of this as using a "lower bound of a lower bound".



(a) Plain Dijkstra. Settled nodes: 107,641, touched edges: 156,641

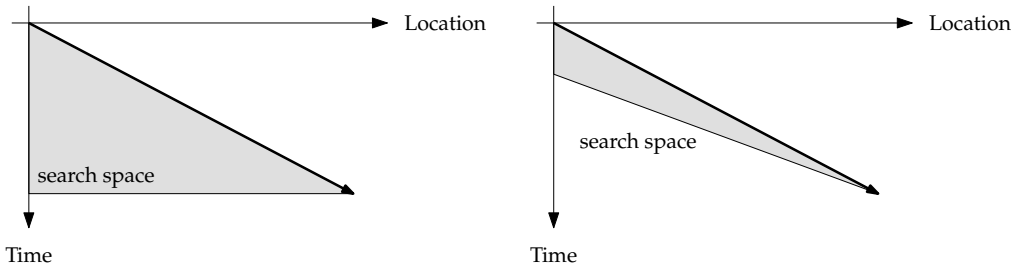
(b) ALT with 16 landmarks, from which 4 have been active during the query. Settled nodes: 17,043, touched edges: 25,238

Figure 26: A query from Straubing to Remscheid-Lennep using plain Dijkstra and the ALT algorithm. The diamond nodes are landmarks whereas landmarks which were active at some time during the query are colored in red.

figure we visualize this by the gray area which is enclosed by the arrow representing the path of the optimal connection through time and space. With plain Dijkstra all nodes are settled which can be reached in time before we reach our target, thus the gray area is a perpendicular triangle. The ALT algorithm however, only touches nodes $v \in V$ with $\text{dist}(v) \leq \text{dist}(t) - \pi(v)$, thus the gray area—and for such the number of settled nodes—is a much smaller strip, since paths to nodes which can not be reached "in time" (regarding the lower bound function π) are cut off during the search.

If we think this further, we can derive the following semantics from this context. Since we prefer nodes in the priority queue, which have a smaller distance from the target, connections which go closer to the target are preferred over connections which only cover a short distance toward the target. These connections are exactly long distance train connections, so the ALT algorithm implicitly prefers long distance trains over short distance trains (as far as they are available), which is the same approach one would use when finding an itinerary by hand.

If we put it into simple words, we can say that Arc-Flags optimizes the geographic dimension, and ALT optimizes the time dimension. This leads us to the question how these two



(a) Plain Dijkstra **(b)** ALT algorithm

Figure 27: Illustrating the differences in search space between a normal Dijkstra and the ALT algorithm: ALT does not crawl so deeply in time, because nodes which are closer to the target (in the sence of travel time) are preferred over nodes from which it takes longer to the target.

speed-up techniques can be combined into one new speed-up technique, and how they perform together. From road networks we know that combining Arc-Flags with ALT does not yield much improvement over using them alone [Sch08]. Mostly, Arc-Flags overrules ALT, and since in road networks there is no time component ALT does not have the advantage of optimizing the time dimension. However, we see in the next section, that in the expanded railway model the two techniques perform very well together because they optimize almost orthogonal leading to speed-ups by the factor of the product of the individual speed-ups of the two techniques.

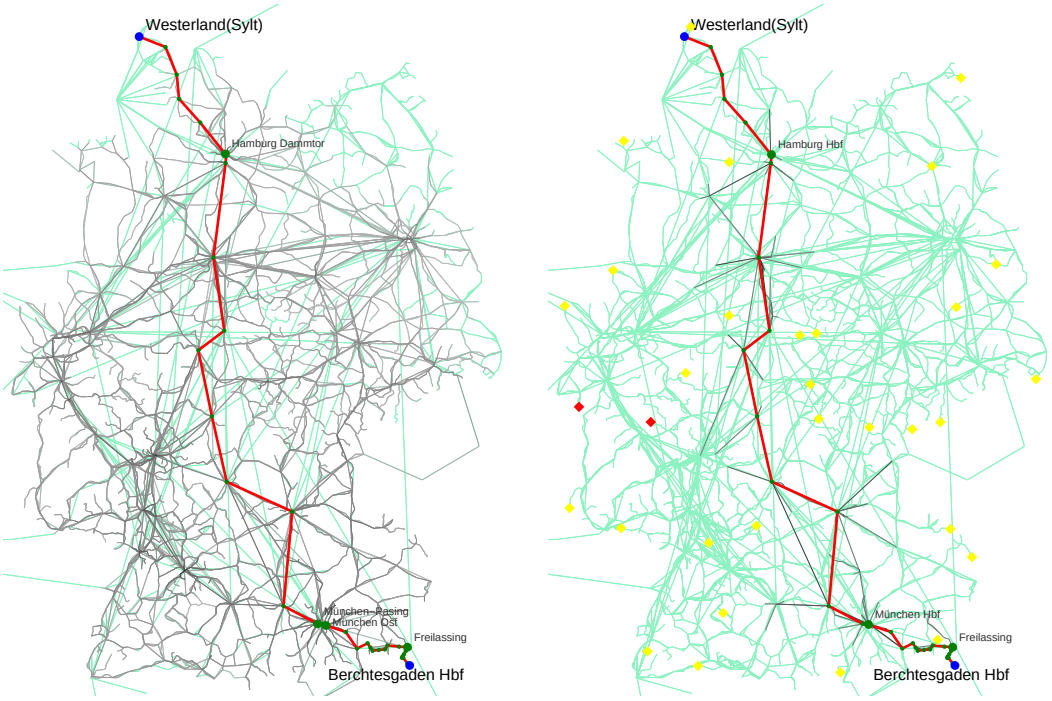
5.3 Combining Unidirectional ALT with Arc-Flags

Combining Arc-Flags with ALT is pretty straight forward, since the two techniques do not interfere with each other. Arc-Flags simply ignores edges with its appropriate flag not being set, and ALT just modifies the key value of the nodes in the priority queue, and thus the order in which they are dequeued. Both modifications can be applied simultaneously without side effects.

Figure 28 shows a sample query from Berchtesgaden in Bavaria to Westerland on Sylt. The difference in the number of settled node and accordingly touched edges is really dramatic.[20] Geographically almost no routes to stations which are not on the optimal path are searched which is due to the Arc-Flags technique, while ALT cuts off the search space in the time dimension, particularly where long distance trains are available. Thus, almost only nodes along the actual shortest path are touched.

In the next section we deal with experiments we have conducted on several graphs. First, we describe the hardware and software which was used in our experiments. After talking about the raw data, and how we convert it to the actual railway model, we evaluate the results we achieved. We see, that the performance of Arc-Flags mainly depends on the right choice of

[20] The example shown is an instance which leads extremely good results. In average, the speedups are not *this* good. A more detailed evaluation can be found in Section 6.



(a) Plain Dijkstra. Settled nodes: 333,851, touched edges: 481,183

(b) Combination of ALT and Arc-Flags with 32 landmarks and 128 regions. The Arc-Flags `decideSameKey` strategy is "geographic distance to target". Settled nodes: 3,029, touched edges: 3,751

Figure 28: A query from Berchtesgaden to Sylt using plain Dijkstra and the combination of ALT and Arc-Flags.

the `decideSameKey` strategy during preprocessing. Further, we show that the combination of ALT with Arc-Flags has a dramatic increase on the speed-up factor over the speed-up factors of the individual techniques.

6 Experimental Studies

In this section we conduct several experiments on the railway models with different speed-up techniques on timetable graphs. Our implementation of both the railway models / graphs and the query techniques are done in C++ solely based on the STL. For efficiency reasons, we do not use virtual methods and class inheritance but rather make excessive use of templates. The way we store graphs is done using a binary format consisting of a *forward star* (adjacency array) representation which is very efficiently implemented [Del07]. For this reason, we can handle huge graphs very well with a minimum of required memory—the performance of reading a graph from file is limited by the hard drive speed. The implementation of our graphs also

```
*Z 27988 RD____    01                                  % 27988 RD____    01
*G DNR 8000247 5400004                                 % 27988 RD____    01
*A VE 8000247 5400004 000000                           % 27988 RD____    01
*A FB 8000247 5400004                                  % 27988 RD____    01
*A RD 8000247 5400004                                  % 27988 RD____    01
*A GR 8005352 5400004                                  % 27988 RD____    01
*GR 8005353 8005352 5400004 Cheb(Gr)                   % 27988 RD____    01
8000247 Marktredwitz              2112                 % 27988 RD____    01
8000613 Arzberg(Oberfr)      2119 2120                 % 27988 RD____    01
8005352 Schirnding           2124 2126                 % 27988 RD____    01
5400004 Cheb                 2140                      % 27988 RD____    01
```

Figure 29: Sample set of raw data belonging to a local train from Marktredwitz to Cheb at the German-Czech border.

contains backward-edges, hence the backward graph $\bar{G}$ is stored implicitly.[21] The choice of the tuple of speed-up technique, `decideSameKey` strategy and model type is based on template arguments, and can be set from the command line when stating queries. For debugging and visualization purposes we also developed an interactive program which allows us to generate itineraries and pictures. As compiler we use GCC version 4.1 on SuSe Linux 10.1 (kernel version 2.6.16.13-4-smp) with the flags `-O4 -DNDEBUG -funroll-loops`.

Our experiments were conducted on a dual core AMD Opteron 2218 processor having 2.6 GHz, 1 MiB level 2 cache (each core) and 16 GiB of main memory. All of our programs are single threaded, and thus only one of the two cores is used.

6.1 Raw Data Conversion

For input we only use real world data which was kindly given to us by HaCon [HIm] (and is also used by the Deutsche Bahn). The raw data first needs to be converted into a set of files representing the simple or realistic time expanded model. Since the format of the raw data is quite unhandily for our purposes, in a first step we convert the raw data into an intermediate format from which we generate the model and graph files. Figure 29 shows an example of a train represented in the raw data files. The first lines consist of some meta data information about the train from which only the train type (in the `*G` line) is relevant for us. Next, for each station the train is passing through, one line exists where the first column is the station's id, the second its name and the third and fourth are the arrival and departure time, respectively. Additionally, a separate file exists where all stations with their id number and their x/y coordinates are listed.

Since our definition of a timetable is based on connections and not on stations being passed through, the intermediate file format for connections is simply a text file, where each line consists of an elementary connections composed of a connection ID, the departure and arrival

[21] If for two nodes $u, v \in V$ both (u, v) and (v, u) are contained in the edge set, the forward and backward edges are *compressed*, i.e. only stored once.

stations/times, the train type and its id. Another intermediate file to represent all the stations is generated where each line belongs to one station composed of its id number, the transfer time of the station, its x/y coordinates and the name. Since the task of converting the raw data into our intermediate format involves heavy parsing and processing of text we use a PERL script for that. The transfer times are generated at random in a specified interval. For our experiments we used a range of 3–7 minutes. In reality one would of course define the transfer time according to the size and topology of the particular station. While the first can be collected from our data, the latter is not available.

In a second step the simple or expanded model is generated from the intermediate data. This is done using C++ as described above, and results in a set of files consisting of two binary graph files for the condensed and expanded graphs, and several (also binary) files needed for the meta information. Furthermore, in this step the radius in kilometers can be specified for which two stations should become neighbours.[22] This set of files is then used as input for our specially developed shortest path query programs.

6.2 Input Graphs

Our raw real world data consists of timetables from the winter period 1995/1996 up to the winter period 2001/2002. From this repertory we decided to choose the following selection for generating input graphs:

- Germany's railway timetable from the winter period 2001/2002—referred to as *de_fern*,
- the railway network of France (including high speed TGV trains) from the winter period 1996/1997—referred to as *fra*,
- the railway network of central Europe from the winter period 1996/1997—referred to as *europe*, and
- the bus and tram network of Berlin from the winter period 2001/2002—referred to as *bvb*.

All experiments were conducted only using the realistic model as described in Section 3.4, since the realistic model has a similar structure as the simple model, and therefore the speed-up techniques should behave in a similar manner as well. Further, we think, that the simple model is too far from realism, because transfers can not be modeled with it, and as a consequence transfer times at stations are ignored which may lead to itineraries that are simply not practical. Table 1 gives an overview about the size of the graphs generated for each instance. Having a look at figures 2 and 30 we see, that the graphs also differ in structure. While the German network is a symbiosis of long and short distance connections which mostly share the same tracks, the French graph mostly consists of short distance connections added by some TGV connections all originating from Paris. In contrast, the graph of Berlin's bus network is drawn

[22] Only if no elementary connections exist between them, see Section 3.5.

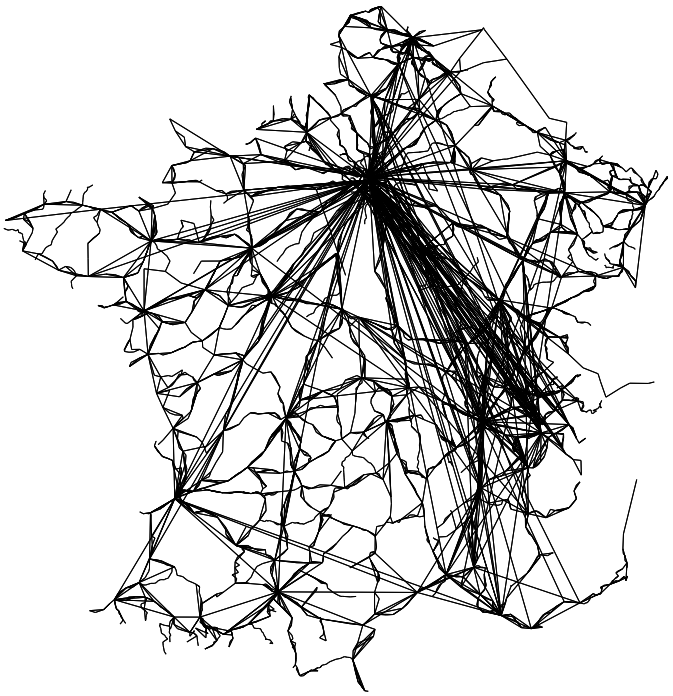


Figure 30: Condensed Graph of France. Note the high speed train (TGV) connections all originating from Paris.

almost planar, and it only consists of "local" connections which run very frequently in part.[23] The Europe graph is our largest instance and is a mix of many different structures of each country. We use this to get some realistic values regarding query time, since today's railway companies mostly do not only have the data of their own country but of other countries as well.

	Exp. Graph		Cond. Graph	
	#nodes	#edges	#nodes	#edges
de_fern	1,661,828	2,731,275	6,730	19,088
fra	616,514	1,014,104	4,551	15,802
europe	5,251,194	8,619,281	29,770	91,586
bvb	2,232,015	3,712,092	2,874	7,530

Table 1: Size of our input graphs.

For our computations, from each of the graphs we only use the largest *strongly connected component* which is extracted from the expanded graph. The condensed graph is then rebuilt according to the stations which are left in the strongly connected component. All other edges and nodes are deleted. This assures that a shortest path between two arbitrary nodes s and t

[23] Some buses are running less frequently, while other are running up to every five minutes or so.

can always be found. The number of nodes and edges of both the expanded and the condensed graphs are shown in Table 1. Please note the tremendous increase when going from the condensed to the expanded graphs which is due to "rolling out" time.[24] Together with the set of meta information files (like the station names, train ids, etc) this results in an altogether file size between 100 and 300 MB per instance.

6.3 Results and Evaluation

In this section we report results we gained in experiments on the above mentioned instances using Arc-Flags and ALT (and the combination of both speed-up techniques). We first focus on Arc-Flags and tend to the ALT algorithm later. All the following tables are made up as follows. For each graph used in the specific experiment we have two groups of columns. The first two columns refer to values regarding the preprocessing phase, namely the preprocessing time and the number of additional bytes per node required to store the preprocessed information. The second group refers to the values obtained from the queries with the specific speed-up technique on the given graph. The number of settled nodes is a direct measure to quantify the size of the search space of a DIJKSTRA run, which we want to minimize of course. Additionally the query time gives some insight on the "real" speed of the algorithm, but may vary depending on the computer hardware used. Additionally in each table the first row of data named "reference" refers to the values obtained by a pure DIJKSTRA run without any speed-up technique. Each value in the "query" group is obtained by running 1.000 queries between uniformly distributed random stations at a random start time and using the average value as the result.

DecideSameKey Strategy

As we worked out in Section 5.1, there are several tuning parameters for Arc-Flags. We start with the most important factor first: The right choice of the implementation of the `decideSameKey` operation during the preprocessing phase. Table 2 compares our four implementations of `decideSameKey`. On all queries we used a partition with 64 regions generated by SCOTCH [Pel07] which was computed on the condensed graph.

We observe that the standard strategy, which works very well on road networks (see [BD08]) totally fails on timetable graphs. Obviously only on very few (almost none) edges the flags are set to zero, so still a lot of unnecessary nodes are explored during the query. Taking the overhead during the query into account, this even leads to "speed-downs" concerning query time. This has two reasons: The enormous number of shortest paths of equal length and the fact that the boundary nodes of the target region have all kind of different timestamps. Regarding the first point, minimizing the number of hops does not lead to a predictable outcome of the `decideSameKey` decision, thus chances are very high, that for two equal shortest paths both paths are opened at some point, which does not give any speed-ups when stating a query due to the tremendous number of shortest paths. The second point is a general drawback of using

[24] In the time dependent models the graph size would correlate to the size of our condensed graphs (q.v. Section 3.2).

	de_fern				*bvb*			
	Prepro		Query		Prepro		Query	
Strategy	[h:m]	[B/n]	#settled	[ms]	[h:m]	[B/n]	#settled	[ms]
reference	—	0	152,998	58.13	—	0	150,460	50.06
hops	17:00	26.2	149,931	70.31	36:30	26.6	146,432	60.68
transfers	16:26	26.2	152,307	71.674	35:34	26.6	149,921	62.28
distance	20:53	26.2	134,462	61.802	36:50	26.6	147,190	61.13
geo. dist. to target	16:08	26.2	38,511	14.96	36:17	26.6	65,184	25.29

Table 2: Arc-Flags. Evaluation of `decideSameKey` strategy. For each strategy we used a partition with 64 regions to generate the arc flags.

the condensed graph as the source for partitioning the expanded graph. Since the boundary nodes consist of all possible times, a shortest path, which is not used for a boundary node with time t_1 is set to `false`, but is overwritten with `true` by some other path that is used for a boundary node with time t_2. For example, we can assume that all edges between transfer nodes are set to `true`. Ultimately, all paths for all time segments are opened, thus yielding no advantage for the actual query when the exact time is known and paths belonging to other time segments should not need to be considered.

Also minimizing transfers or the accumulated travel distance still gives too chaotic results of the `decideSameKey` operation. The results are even worse. Only using the minimum of the direct geographic distance from the two involved nodes to the target node as a criterion for the `decideSameKey` operation forces some edges being set to `false` because this almost always forces the same outcome of the `decideSameKey` operation. The downside of this approach is, that we only involve geographical information and thus lose the opportunity of optimizing in the time dimension completely.

If we look at the time we need to preprocess the data, the results of the first three strategies are even more scary. The very high value of the distance strategy comes from the massive amount of floating point arithmetic used to calculate the accumulated distance along the paths in the Dijkstra search.

Number of Regions

Since the geographical distance strategy is the only one which gives some noteworthy results, we use it in all subsequent experiments. Table 3 shows the next tuning parameter for Arc-Flags we examined: The number of regions of the partition.

As one would expect, increasing the number of regions leads to better speed-ups, since more paths can be cut off. However, doubling the number of regions does not lead to halving the number of settled nodes. Furthermore, the railway graph performs about as twice as good as the bus network graph. However, we can observe that the query time directly correlates to the number of settled nodes which is due to the negligible overhead of Arc-Flags during the query when increasing the number of regions. With both graphs the difference in query time between 64 and 128 regions is extremely small compared to the increase in preprocessing time from e.g. 16:08 hours to 24:14 hours in the de_fern graph, so it probably makes sense to

	de_fern				*bvb*			
	Prepro		Query		Prepro		Query	
#regions	[h:m]	[B/n]	#settled	[ms]	[h:m]	[B/n]	#settled	[ms]
reference	—	0	152,998	58.13	—	0	150,460	50.06
8	3:46	3.3	64,848	27.05	9:30	3.3	96,047	37.08
16	5:00	6.6	53,081	21.56	17:54	6.7	86,388	33.17
32	12:05	13.1	44,651	17.68	29:34	13.3	77,595	29.73
64	16:08	26.2	38,511	14.96	36:17	26.6	65,184	25.29
128	24:14	52.6	34,894	13.45	50:35	53.2	60,101	22.60

Table 3: Arc-Flags. Testing the impact of increasing the number of regions using the "geographic distance to target" strategy for `decideSameKey` during preprocessing.

use 64 regions at most. Generally, we always have to make a trade-off between an increase in preprocessing time and as a result a decrease in query time.

ALT

The next experiment is devoted to the ALT algorithm. As we explained in Section 5.2, we generate landmarks using the MaxCover method which is described in [DSSW06]. Table 4 shows the influence of the number of landmarks on the query time and the number of settled nodes.

	de_fern				*bvb*			
	Prepro		Query		Prepro		Query	
#landmarks	[sec]	[B/n]	#settled	[ms]	[sec]	[B/n]	#settled	[ms]
reference	—	0	152,998	58.13	—	0	150,460	50.06
2	< 1	0.06	70,246	30.85	< 1	0.02	122,598	48.80
4	< 1	0.13	62,484	28.92	< 1	0.04	117,937	48.46
8	1	0.26	59,039	28.07	< 1	0.08	116,988	50.26
16	1	0.52	57,898	28.53	1	0.16	116,359	50.96
32	5	1.04	57,424	29.58	2	0.33	116,093	51.60
64	13	2.07	56,337	29.88	5	0.66	115,660	51.74

Table 4: ALT. Testing the effect of using different numbers of landmarks.

First of all, the preprocessing time is really negligible (compared to Arc-Flags), which makes this speed-up technique also interesting for dynamic scenarios where the graph may change and therefore a fast re-computation of the preprocessed data is required. Second, the additional space required is also very small, since we calculate landmarks on the condensed graph, and thus the distance table from all nodes to each landmark is small as well. The downside of ALT is, however, that the computational overhead during the queries is more relevant. As we see at the de_fern graph, an increase from 8 landmarks on only leads to insignificant changes in the number of settled nodes, whereas the query time starts to rise again, which is due to the increasing effort for examining more potential candidates for active landmarks during the query.

If we look at the bvb graph, the ALT algorithm does not seem to draw any distinction compared to pure Dijkstra. With the structure of the local bus network graph having no long distance connections, the ALT algorithm has almost no margin of optimization with the key in the priority queue.

Combining ALT and Arc-Flags

The next Table 5 reports the performance of each speed-up technique on all four input instances.

	de_fern				*bvb*			
	Prepro		Query		Prepro		Query	
Algorithm	[h:m]	[B/n]	#settled	[ms]	[h:m]	[B/n]	#settled	[ms]
Plain Dijkstra	—	0	152,998	58.13	—	0	150,460	50.06
Arc-Flags	24:14	52.6	34,894	13.44	50:35	53.2	60,101	23.09
Uni-ALT	1 s	0.26	59,039	28.77	< 1 s	0.08	116,988	50.28
Arc-Flags w/ ALT	24:14	52.86	14,790	6.73	50:35	53.3	48,699	22.16

	fra				*europe*			
	Prepro		Query		Prepro		Query	
Algorithm	[h:m]	[B/n]	#settled	[ms]	[h:m]	[B/n]	#settled	[ms]
Plain Dijkstra	—	0	113,167	37.77	—	0	1,113,707	585.20
Arc-Flags	3:38	52.6	35,168	12.89	116:13	52.2	187,489	91.04
Uni-ALT	3 s	0.47	55,356	23.42	2 s	0.36	443,410	281.78
Arc-Flags w/ ALT	3:38	53.1	16,779	7.30	116:13	52.6	63,561	33.60

Table 5: Comparing the two speed-up techniques and their combination. The parameter set used is 128 regions, "geographical distance to target" `decideSameKey` strategy for Arc-Flags and 8 landmarks using MaxCover for ALT.

As opposed to road networks, on all four graphs ALT and Arc-Flags behave orthogonal as the speed-up factors approximately multiply when the two techniques are combined. The best results are obtained on the Europe graph where (regarding query time) Arc-Flags leads to a speed-up $T_{\text{Dijkstra}}/T_{\text{technique}}$ of 6.4, ALT to a speed-up of 2.1 and the combination of both techniques to a speed-up of 17.4 which is even beyond the product. The worst speed-ups can be observed on the homogeneous bus network graph bvb which seems to be a very hard instance.

Local Queries

In all of our previous tables we created random queries by choosing a random start and target station as well as a completely random start time. Of course these queries are not very realistic, since in reality most people will not want to travel over night, but when creating purely random queries this matter is not accounted for at all. Hence, we now assume that the longer the distance put back the earlier the starting time should be chosen. Further, we want to examine the relation between the distance of the start and target station and the performance of the speed-up technique.

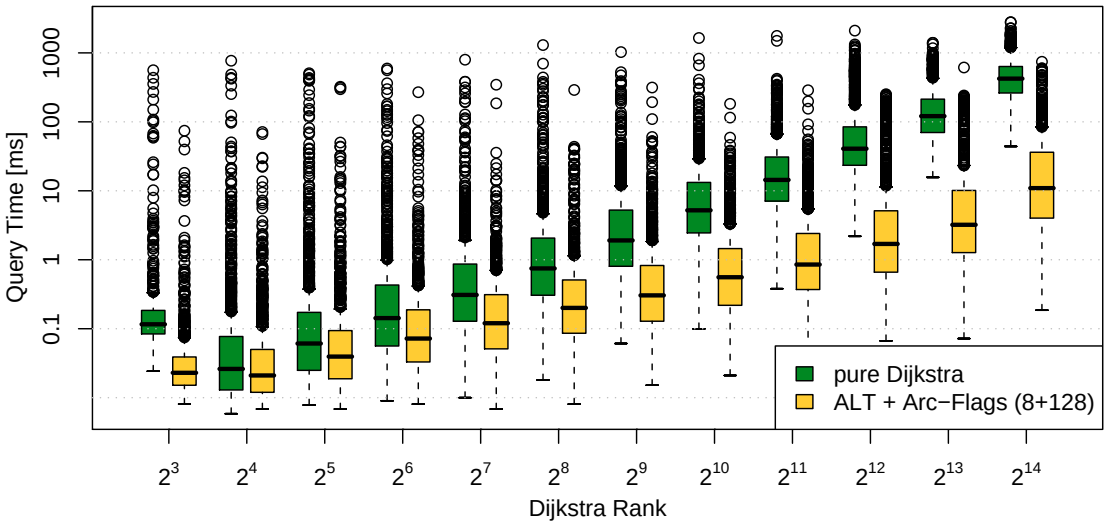


Figure 31: Comparison of Dijkstra with ALT and Arc-Flags using Dijkstra rank methodology from [SS05] on the graph of Europe by a box and whisker plot [Tea04]. Each box spreads from the lower to the upper quartile and contains the median. The whiskers extend to the minimum and maximum value omitting outliers, which are plotted individually.

For this reason, we have used the Dijkstra rank methodology described in [SS05] to visualize the performance of the combination of ALT and Arc-Flags compared to the Dijkstra ranks as shown in Figure 31. The Dijkstra rank is defined as follows. If we order the nodes V of the given graph as they are dequeued (settled) by the priority queue of Dijkstra's algorithm, then the Dijkstra rank $r(v)$ of a node v is defined as the rank of v with respect to this order. The source node s has rank $r(s) = 0$, its first considered neighbour rank 1 and so on. For each Dijkstra rank between 2^3 and 2^{14} we have created 1000 random queries on the condensed graph of Europe to determine the source and target stations which are then used as input for queries on the expanded graph. The start times are not chosen by random but are always around 7:00 in the morning for simplicity. This should guarantee for most queries that they can be accomplished on one day without the need for an overnight journey. We have decided to use the graph of Europe for this experiment, since it provides a good mixture of long and short distance trains.

As we see in Figure 31 using ALT with Arc-Flags benefits more from long distance trains. In short queries with low Dijkstra rank the advantage of the speed-up technique to a pure Dijkstra query is rather negligible, but the longer the source and target stations are apart the lower are the query times of the speed-up technique compared to the pure Dijkstra implementation. Please note that both axis are scaled logarithmically, hence our speed-up technique scales linear with increasing Dijkstra rank but with a much flatter curve than pure Dijkstra.

Concluding, we showed that executed on their own, neither technique yields very good speed-ups, although Arc-Flags exceeds the ALT algorithm regarding query time which comes

at the price of preprocessing time. On the other hand, ALT can be applied with almost no overhead regarding preprocessing and query time. However, when combined, Arc-Flags and ALT perform very well outperforming all known speed-up techniques for time expanded networks [PSWZ07].

7 Conclusion and Outlook

In this work we investigated several approaches how timetable data can be modeled as graphs to solve the *earliest arrival problem* of finding an optimal itinerary. From these approaches the *realistic model* is the most realistic version when considering the *time expanded* approach. We showed that with little effort DIJKSTRA's algorithm can be adapted to solve the earliest arrival problem on this model.

Because of the huge size of the graphs with several million nodes a pure DIJKSTRA query is not very effective. Unfortunately, the speed-up techniques which were mainly developed for road networks, and have been highly optimized on those, can not simply be used on timetable data without further consideration. First, hierarchical techniques use bidirectional search, rendering them useless on our models, since the target node is not known in advance. For that reason, we focused on two goal directed unidirectional techniques, namely Arc-Flags and ALT, and showed how they must be adapted to work with our models. Second, we have worked out that due to the tremendous number of shortest paths of equal length between two nodes, the choice of the right implementation of the `decideSameKey` operation in the preprocessing phase is crucial. Minimizing hops, which is obviously a good choice on road networks, performs extremely bad here. But also other strategies, like minimizing transfers or the accumulated travel distance, which intuitively seem to be good at first, do not perform well. Only the most aggressive strategy, geographically minimizing the distance to the target station, leads to notable speed-ups.

Moreover, we worked out, that Arc-Flags and ALT optimize in two different ways. Arc-Flags (with the geographic distance strategy) prunes paths which lead to the wrong direction geographically, while the ALT algorithm "cuts off" paths which do not lead to the target station fast enough, thus having the sense of preferring fast trains over slow trains or in other words optimizing the time component. This lead us to the conclusion that—opposed to on road networks—the combination of Arc-Flags and ALT gives an extra boost in query time because both of the aspects mentioned above behave orthogonally.

Our experiments, which were conducted only using real world data and realistic queries, revealed that with a good choice regarding the `decideSameKey` operation, Arc-Flags generally yields good speed-ups. The performance of the ALT algorithm in contrast heavily depends on the structure of the underlying timetable network. On Germany's railway network ALT performs very well, because it can play off its advantage of preferring long distance trains. On the bus network of Berlin however, the speed-ups are negligible since there are no long distance connections at all. But it should be mentioned that the ALT algorithm with its negligible preprocessing time can always be applied without yielding any disadvantages.

The best results were obtained using the combination of ALT and Arc-Flags which increased the speed-ups up on all tested instances approximately by the product of the individual speed-ups of each technique. On the graph of whole Europe, this even lead to a average speed-up of 17.4 regarding query time.

Summarizing, we showed that two of the most prominent goal directed speed-up techniques can be adapted to timetable networks and perform very well. However, when using techniques derived from road networks a lot of consideration has to be put into the adaption process. With speed-ups up to 17.4 gained by the combination of ALT with Arc-Flags, we have achieved the best known results for real world queries on time expanded graphs.

Future Work

The field of developing good speed-up techniques for timetable networks is still widely unexplored. As we showed with Arc-Flags, just copying speed-up techniques from road networks over to timetable networks can lead to a big disillusion as the speed-ups can become insignificant, although on road networks they seemed very promising. However, with some further engineering and fine-tuning these techniques eventually provide good results.

The speed-up techniques discussed in this work certainly are not the end of the line. We would be interested in further strategies for the `decideSameKey` operation during the preprocessing of the Arc-Flags, as we believe this is the main point where we still have room for further improvements concerning basic Arc-Flags. Our best strategy, geographic distance, does not exploit the additional information provided by the expanded graph over the condensed graph, since the geographic distance between stations is already available from the condensed graph.

Moreover, Bauer and Delling have presented in [BD08] an improved version of Arc-Flags, named SHARC, which in a first step contracts the graph before computing the Arc-Flags on the new, smaller graph. This does not only speed-up the preprocessing time—which is too large for practical use in our cases—but also decreases query times significantly, making it the fastest unidirectional speed-up technique (on road networks) available. We are very optimistic that some of these techniques can be adapted to timetable networks and equally lead to an increase of the speed-up factor.

Considering the ALT algorithm, it would be interesting to see whether we can further improve the quality of the lower bounds. Since landmarks are computed on the condensed graph, which is pretty small, we could imagine using the target station as an “optimal” landmark and compute the distance table to the target node on the fly for each query.

Furthermore, the exploration of dynamic scenarios, like delay management, would be interesting. On one hand, the models probably need to be adapted to cope with dynamic scenarios, and on the other hand, the preprocessed data from the speed-up techniques needs to be dynamically updated. This reveals the biggest disadvantage of the Arc-Flags method because the preprocessing takes too much time, and can not be updated easily. In this case the ALT algorithm may be the stronger choice.

As long term research we could imagine some new speed-up techniques to arise that do not come from road networks but are optimized for timetable graphs. As an example, when we

try to compute a good itinerary by hand, we do it probably with a hierarchical approach by looking for a local train connecting to the nearest station which provides high speed trains, then using these high speed trains to go as close to the target as possible, and do the last step by local trains again. Although the ALT algorithm implicitly benefits from this property on most networks, one could think of a speed-up technique which exploits this property more explicitly. Another interesting characteristic to exploit is the fact that most modern timetables are synchronized ones, i.e. trains depart every 10, 30 or 60 minutes. This is especially true for local networks like bus or commuter train networks.

Unfortunately, speed-ups in the scale of 17 stand in no correlation to the increase in graph size when constructing the expanded graph from the condensed graph which is a factor of about 247 in case of the de_fern graph regarding the number of nodes. Hence, a pure DIJKSTRA query on the time dependent model is still faster than using the expanded model with one of our speed-up techniques. Furthermore, dynamic scenarios like delays can be handled easier on the time dependent model, see [DGWZ08]. For that reason, the time dependent approach may be the more seminal one, thus we would be interested in future work on the time dependent model, including improvements regarding the models as well as speed-up techniques.

Acknowledgments

First of all, I would like to thank Prof. Dr. Dorothea Wagner for giving me the opportunity to work on this highly interesting topic; Daniel Delling and Rheinhard Bauer for giving me support and the plenty of time we spent having discussions which were always very inspiring. They were always available and helpful when I had a question or problem. Furthermore, I would like to thank my friends who helped me with the many little things, especially Christoph Klee for some interesting discussions about the speed-up techniques and Julian Hörst for helping me out with C++ when I got stuck. Finally, I would like to thank my parents who have supported me through my whole studies and without whom I would not have had the chance to write this thesis.

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig angefertigt habe und nur die angegebenen Hilfsmittel und Quellen verwendet wurden.

Karlsruhe, den 31. Januar 2008

References

[BD08] Reinhard Bauer and Daniel Delling. Sharc: Fast and robust unidirectional routing. In *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX'08)*, pages 13–26. SIAM, 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)*. Schloss Dagstuhl, Germany, 2007.

[BFSS07] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. Fast routing in road networks with transit nodes. *Science*, 316(5824):566, 2007.

[BJ04] Gerth Brodal and Riko Jacob. Time-dependent networks as models to achieve fast exact time-table queries. In *Proceedings of ATMOS Workshop 2003*, pages 3–15, 2004.

[CH66] K. Cooke and E. Halsey. The shortest route through a network with time-dependent intermodal transit times. *Journal of Mathematical Analysis and Applications*, (14):493–498, 1966.

[Del07] Daniel Delling. A fast framework for engineering shortest path algorithms. 2007.

[Dem07] Camil Demetrescu, editor. *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*. Springer, June 2007.

[DGJ06] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors. *9th DIMACS Implementation Challenge - Shortest Paths*, November 2006.

[DGWZ08] Daniel Delling, Kalliopi Giannakopoulou, Dorothea Wagner, and Christos Zaroliagis. Timetable information updating in case of delays. Technical report, Arrival Technical Report, 2008.

[Dij59] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[DSSW06] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Highway hierarchies star. In Demetrescu et al. [DGJ06].

[DW07] Daniel Delling and Dorothea Wagner. Landmark-based routing in dynamic graphs. In Demetrescu [Dem07], pages 52–65.

[GH04] Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: A* search meets graph theory. Technical Report MSR-TR-200, Microsoft Research, 2004.

[Hil07] Moritz Hilger. Accelerating point-to-point shortest path computations in large scale networks. Master's thesis, Technische Universität Berlin, 2007.

[HIm] Germany HaCon Ingenieurgesellschaft mbH, Hannover. Hafas, a timetable information system.

[HKMS06] Moritz Hilger, Ekkehard Köhler, Rolf Möhring, and Heiko Schilling. Fast point-to-point shortest path computations with arc-flags. In Demetrescu et al. [DGJ06].

[HNR68] Peter E. Hart, Nils Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.

[KMS05] Ekkehard Köhler, Rolf Möhring, and Heiko Schilling. Acceleration of shortest path and constrained shortest path computation. In WEA'05 [WEA05], pages 126–138.

[Lau97] Ulrich Lauther. Slow preprocessing of graphs for extremely fast shortest path calculations, 1997. Lecture at the Workshop on Computational Integer Programming at ZIB.

[Lau04] Ulrich Lauther. An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background. volume 22, pages 219–230. IfGI prints, 2004.

[MHSWZ07] Matthias Müller-Hannemann, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Timetable information: Models and algorithms. In *Proceedings of the 4th Workshop on Algorithmic Methods for Railway Optimization (ATMOS'04)*, volume 4359 of *Lecture Notes in Computer Science*, pages 67–90. Springer, 2007.

[MSS+05] Rolf Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning graphs to speed up dijkstra's algorithm. In WEA'05 [WEA05], pages 189–202.

[MSS+06] Rolf Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning graphs to speedup dijkstra's algorithm. *ACM Journal of Experimental Algorithmics*, 11:2.8, 2006.

[OR90] Ariel Orda and Raphael Rom. Shortest-path and minimum delay algorithms in networks with time-dependent edge-length. *Journal of the ACM*, 37(3):607–625, 1990.

[Pel07] Francois Pellegrini. Scotch: Static mapping, graph, mesh and hypergraph partitioning, and parallel and sequential sparse matrix ordering package, 2007.

[PSWZ07] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient models for timetable information in public transportation systems. *ACM Journal of Experimental Algorithmics*, 12:Article 2.4, 2007.

[Sch05] Frank Schulz. *Timetable Information and Shortest Paths*. PhD thesis, 2005.

[Sch07] Dominik Schultes. *Route Planning in Road Networks*. PhD thesis, 2007. submitted.

[Sch08] Dennis Schieferdecker. Systematic combination of speed-up techniques for exact shortest-path queries. Master's thesis, Institut für Theoretische Informatik - Universität Karlsruhe (TH), January 2008.

[SS05] Peter Sanders and Dominik Schultes. Highway hierarchies hasten exact shortest path queries. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05)*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2005.

[SS07] Peter Sanders and Dominik Schultes. Engineering fast route planning algorithms. In Demetrescu [Dem07], pages 23–36.

[SWW99] Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's algorithm on-line: An empirical case study from public railroad transport. In *Proceedings of the 3rd International Workshop on Algorithm Engineering (WAE'99)*, volume 1668 of *Lecture Notes in Computer Science*, pages 110–123. Springer, 1999.

[Tea04] R Development Team. R: A language and environment for statistical computing, 2004.

[WEA05] *Proceedings of the 4th Workshop on Experimental Algorithms (WEA'05)*, Lecture Notes in Computer Science. Springer, 2005.

[WW07] Dorothea Wagner and Thomas Willhalm. Speed-up techniques for shortest-path computations. In *Proceedings of the 24th International Symposium on Theoretical Aspects of Computer Science (STACS'07)*, volume 4393 of *Lecture Notes in Computer Science*, pages 23–36. Springer, 2007.