Time-Dependent Route Planning with Contraction Hierarchies

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Gernot Veit Eberhard Batz

aus Karlsruhe

Tag der mündlichen Prüfung:	8. Juli 2014
Erster Gutachter:	Prof. Dr. Peter Sanders
Zweiter Gutachter:	Prof. Dr. Matthias Müller-Hannemann

Beauty is the splendor of truth.

Plato

Abstract

Route planning algorithms have gained considerable interest in the last ten years. Contraction hierarchies (CHs) [44] are a well-known algorithmic framework for fast and exact route planning on road networks with constant travel costs. Like most other route planning techniques, CHs enable fast *querying* at the cost of a slow *preprocessing*. This thesis generalizes CHs to work with *time-dependent travel costs*. Considered kinds of time-dependent travel costs are *travel times* as well as travel times with *additional time-invariant costs*. The travel times must fulfill the *FIFO property*; that is, a later departure never results in an earlier arrival.

A possible application of time-dependent travel times is to avoid regular congestions and rush hours extracted from statistically collected data. Additional time-invariant costs only seem to be a slight generalization on the first glance, but make the route computation much more difficult; that is, NP-hard.[1] From the application point of view, additional time-invariant costs can be used to penalize inconvenient roads (e.g., narrow, winding, steep, or dangerous). Another use case are monetary travel costs that consist of a time-invariant part (e.g., motorway tolls) and a time-dependent part proportional to travel time (e.g., the wages of a truck driver).

Without additional time-invariant costs, we achieve fast and exact time-dependent route planning. The underlying *time-dependent contraction hierarchies (TCHs)*, and its variants *approximate TCHs (ATCHs)* and *inexact TCHs*, are far from being a straightforward extension of CHs. Instead, several sophisticated algorithmic ingredients are necessary to make them run efficiently. The most important concept is the careful use of approximate computations to find small subgraphs that contain all relevant nodes and edges. Exact computations are performed afterwards, only on the small subgraphs. This simple but effective idea

[1]The NP-hardness of time-dependent route planning with additional time-invariant costs follows from the NP-hardness of a very restricted special case [3].

- makes the running time of the TCH preprocessing at all feasible,
- speeds up expensive exact computations without loosing exactness, and
- enables exact computations with space saving approximate data.

For an instance of the German road network with 4.7 million nodes and 10.8 million edges we are able to answer an *earliest arrival query*—that is, a point-to-point query asking for the minimum travel time for a given single departure time—within less than 1.2 msec. For a *travel time profile query*—that is, a point-to-point query asking for a function that yields the minimal possible travel times over an interval of departure times—we need less than 35 msec, if the departure interval has a width of 24 hours. For inexact travel time profile queries with an error of up to 1 % this reduces to 2.6 msec. The underlying ATCHs and inexact TCHs that represent the instance of the German road network all need less than 1 GiB space and can be generated within about 30 minutes. With 8 parallel threads this even reduces to about 5 minutes.

With additional time-invariant costs, we only achieve inexact route planning. The underlying variant of TCHs is a *heuristic TCH* that does not guarantee to provide minimum cost paths for all possible combinations of start, destination, and departure time. In our experiments, however, the error is practically negligible. The answering of a *minimum cost query*—that is, a point-to-point query asking for a minimum cost path for a given single departure time—takes less than 35 msec for the hardest instance considered in our experiments. This is, again, an instance of the German road network, but this time with monetary travel costs where

- the time-invariant part represents motorway tolls as well as a simple approximation of energy costs proportional to the travel distance, and
- the time-dependent part represents an amount of money proportional to the travel time as defined by a given hourly rate.

The preprocessing takes 1 hour and 23 minutes with 8 parallel threads for this set of time-dependent travel costs. But note that the underlying implementation of the preprocessing is prototypical. So, smaller preprocessing times may be possible. Of course, we expect larger preprocessing times than without additional costs, even with a well-tuned implementation.

The resulting heuristic TCH structure needs 9 GiB, but we have not applied approximation to reduce space usage so far (as we do in case of ATCHs and inexact TCHs). The main outcome is that heuristic TCHs allow practically exact and sufficiently fast preprocessing and querying for the tested instances. The computation of *cost profile queries*—that is, of point-to-point queries asking for a function that yields the minimal possible travel costs over an interval of departure times—is only considered as part of the preprocessing. We show, however, that cost profiles have exponential worst case size, which makes them significantly more complex than travel time profiles.[2] An interesting observation is that heuristic TCH structures have to deal with more general TTFs with points of discontinuity and with more general additional costs that are time-dependent. This suggests that heuristic TCHs may work if an original road networks comes with such generalized travel costs (though waiting gets beneficial in this setup).

We not only evaluate our TCH-based route planning techniques experimentally, but provide proofs of correctness for all of them. All these route planning techniques are assembled from a number of basic algorithmic ingredients that are variants of Dijkstra's well-known algorithm. We speak of *Dijkstra-like* algorithms hence. Some of these algorithms compute exact results and some approximate results. We provide proofs of correction for all of these ingredients to build a basis for the proofs of correctness of the TCH-based route planning techniques.

[2]According to Foschini et al. [36], travel time profiles have a worst-case size of $K \cdot n^{\mathrm{O}(\log n)}$, where n is, in principle, the size of the road network without travel time data and K the total size of the travel time data.

Acknowledgements

I would like to thank a number of people. First of all, I want to thank my doctoral advisor Peter Sanders for inspiring and fruitful discussions, his advice and his support. Also, I want thank the other route planning people who are or were at Karlsruhe. Thank you, Robert Geisberger, for many interesting (not only route planning related) discussions and good acquaintance. Thank you, Dominik Schultes, for a lot of help when I started working on route planning. Thank you, Christian Vetter, for creative problem solving during the student research project and as a student assistant. Thank you, Daniel Delling, Julian Dibbelt, Andreas Gemsa, Dennis Luxen, Moritz Kobitzsch, and Dennis Schieferdecker for several fruitful discussions. Also, I want to thank Sabine Neubauer for providing us her implementation of the Imai-Iri algorithm that she produced in her student research project. Further, I want to thank all the coauthors I had in the last years. Their names are a subset of the names already mentioned.

Especially, I want to thank all the office mates that I had during my years at KIT. Thank you, Dominik Schultes, Dennis Luxen, Robert Geisberger, and Timo Bingmann for inspiring discussions (sometimes work-related and sometimes not) and the good atmosphere. I also want to thank all the other people at the groups of Peter Sanders and Dorothea Wagner for the atmosphere; especially, Anja Blancani, who supports us all, Dennis Schieferdecker, for the coffee breaks, and Jochen Speck, for several interesting lunch discussions.

Of course, I want to thank family and friends. Fist, I want to thank my parents for their constant encouragement and support. Further, I want to thank my friends David, Jonathan, Jonathan, Markus, Moritz, Steffi and Daniel, Tobias, as well as several others. Last but not least, I want to thank my lord and savior Jesus Christ. All good things come through him.

❧

Contents

1

Introduction

1.1 Motivation

In the last decade route planning has gained considerable interest of the algorithm research community (see some overview papers [4, 25, 28, 80, 83]). Especially route planning in road networks has really become a success story. The fastest current algorithms compute optimal routes in continental size networks in significantly less than a microsecond [1]. With the availability of highly efficient algorithms, route planning has become popular for practical use: not only for individuals planning car trips but also for companies planning truck tours for example. Web mapping services like Bing Maps, Google Maps, or HERE usually provide route planning capability and have become part of our everyday life.

It is important to know that such a great availability of on-line route planning services would not have been possible without the last decades progress in the development of route planning algorithms. The reason is that Dijkstra's algorithm [33]—once the state-of-the-art method to compute optimal routes in road networks—runs much too slowly to be used on high-throughput servers. On continental size road networks, for example, Dijkstra's algorithm takes up to several seconds. The current algorithmic methods for route planning run several orders of magnitudes faster than Dijkstra's algorithm. The time needed to answer a route planning query should be in the area of milliseconds or below. Times around 0.1 seconds are still tolerable, which is, from the users point of view, virtually at once.

1.1.1 Time-Dependent Route Planning

Most of the progress in route planning algorithms regards to the special case that travel costs are considered as *non-negative constant values*, often highly correlated with travel time. On the one hand, this relatively simple model implies a convenient problem structure that simplifies the design of efficient algorithms a lot. On the other hand, there are real-life factors that can not be handled by constant travel costs. These are, for example,

- travel costs that depend on the time of day,
- knowledge about the uncertainty of travel costs, or
- the need to asses routes with respect to different criteria that can not be compared directly.

As a consequence, several practical aspects—like rush hours, regularly occurring congestions, reliability, or multiple modes of transport—can not be dealt with properly or must be ignored. So, researchers started to work on route planning with more general kinds of travel costs.

In this thesis we focus on *time-dependent* route planning where travel costs depend on departure time. This may be the time of day, a day of the week, or a combination of both. Two types of such *time-dependent travel costs* are considered: *time-dependent travel times* as well as time-dependent travel times with *additional time-invariant costs*. Uncertainty of travel costs and incomparable cost criteria are *not* considered in this thesis. These aspects are subject of *stochastic* and *multi-criteria* route planning respectively—two areas which are considerable challenges on their own.

Time-Dependent Travel Times. Time-dependent travel times are the simplest type of time-dependent travel costs. The edge weights are travel times that depend on the departure time. They are especially suited to model periodically occurring effects like the aforementioned rush hours and regularly occurring congestions. The knowledge about such effects has to be gathered from statistical data. Another possible source of time-dependent travel time data could be traffic simulation based on the current traffic situation. This can be combined with statistically gathered data of course.

Note that time-dependent travel times exhibit a far less convenient problem structure than constant travel costs. This results in increased computational complexity and memory consumption. Attacking these issues requires careful consideration, which makes time-dependent route planning a serious challenge.

Additional Time-Invariant Costs. To obtain a more general type of time-dependent travel costs than time-dependent travel times, we add additional time-invariant (i.e., constant) cost values. This is interesting from the application point of view because dealing only with travel time lets several highly practical aspects unconsidered. Additional time-invariant costs can model, for example,

- energy consumption to avoid large detours that only save a few minutes of travel-time (even non-environmentalists may appreciate that),

- tolls that users want to get rid off if this does not need too much time, or
- penalties to avoid residential areas, inconvenient roads (i.e., narrow, winding, steep, bumpy), or forces of nature (i.e., falling rocks, snow, black ice).

Above we state that time-dependent travel times are more difficult to deal with than constant travel costs. With additional time-invariant costs this is even worse: Route planning gets NP-hard and so far we only have a heuristic route planning algorithm for this setup.

1.1.2 Varieties of User Queries

Our goal is to efficiently answer different varieties of user queries in the context of time-dependent route planning. In route planning we are mainly interested in *one-to-one* queries. This means that a fixed start and a fixed destination are given by the user. In this thesis we distinguish queries with a given *single* departure time from queries with a given departure *interval*. The former are usually referred to as *single departure queries*, the latter as *profile queries*. Also, we distinguish queries where the travel costs are simply the travel times from queries with more general travel costs. Table 1.1 outlines the four resulting varieties of user queries. A single departure query is needed to compute a good route if a fixed departure time is given. A profile query is helpful if the user wants to choose a convenient departure time, for example, to avoid congestions or high travel costs at all.

In route planning, road networks are usually modeled as directed graphs $G = (V, E)$ with some travel cost assigned to each edge. Nodes represent junctions and edges represent road segments for example.[1] Routes in the road network correspond to paths $\langle s \to \cdots \to t \rangle$ in G and vice versa. *Optimal* routes in the road network correspond to paths in G with optimal total travel cost. In case of non-negative constant travel costs we simply assign a value $c_e \in \mathbb{R}_{\geq 0}$ to every edge $e \in E$. In time-dependent route planning, edge weights are not only constant values but functions $C_e : \mathbb{R} \to \mathbb{R}_{\geq 0}$ where $C_e(\tau)$ is the cost of traveling edge e at time τ.

Minimizing Time-Dependent Travel Times. First, we consider the case that the travel cost of a route is the time needed to travel this route. This is a relatively special case of time-dependent route planning. The weight of an edge $e \in E$ is a *travel time function (TTF)* $f_e : \mathbb{R} \to \mathbb{R}_{\geq 0}$; that is, $C_e := f_e$. The TTF f_e yields the amount of time $f_e(\tau)$ that we need to travel along the edge e when departing at time τ. In this thesis all TTFs of edges are continuous piecewise linear functions.

[1]This is not the only possible interpretation of course. Alternatively, nodes could represent road segments and edges could represent transitions between road segment. This *edge-based* model is a possible way to include *turn restrictions* and *turn costs* into route planning [15, 85].

	kind of user query	
cost type	single departure time	departure interval
travel time only	*earliest arrival (EA) query* $O(\lvert V\rvert \log F + \lvert V\rvert \log \lvert V\rvert)$ time (see Section 4.2.1)	*travel time profile (TTP) query* $O(\lvert V\rvert^2 \log \lvert V\rvert) \cdot F \lvert V\rvert^{O(\log \lvert V\rvert)}$ time (from [36], see Section 1.3.2)
additional time-invariant costs	*minimum cost (MC) query* NP-hard (from [3], see Section 6.1.1)	*cost profile (CP) query* NP-hard (from [3], see Section 6.1.1)

Table 1.1. The four different kinds of time-dependent user queries considered in this thesis (printed in italics) and their complexity. If travel cost and travel time are the same ("travel time only"), we have *earliest arrival queries* and *travel time profile queries*. For more general travel costs—that is, travel time with "additional time-invariant costs"—we have *minimum cost queries* and *cost profile queries*. If the user gives a departure interval and not only a single departure time, then we have *profile queries* instead of *single departure queries*. With F we denote the maximum complexity (i.e., number of bend points) of any travel time function in the underlying road network. The underlying road networks are assumed to be sparse (i.e., every node has $O(1)$ edges) with all TTFs fulfilling the FIFO property (i.e., a later departure never results in an earlier arrival).

Also, they all fulfill the *FIFO property* which means that nobody arrives earlier if one starts later. More formally, a TTF f fulfills the FIFO property if $\tau < \tau'$ implies $f(\tau) + \tau \leq f(\tau') + \tau'$. The resulting problem structure is considerably more complicated than in case of constant edge weights. Depending on the fact whether the user gives a single departure time or a departure interval we have two kinds of user queries in this setting: *earliest arrival (EA) queries* and *travel time profile (TTP) queries*.

Earliest Arrival (EA) Query. Given a start node $s \in V$, a destination node $t \in V$, and a departure time $\tau_0 \in \mathbb{R}$ we want to know the earliest possible time for arriving at t as well as a corresponding path in G.

Travel Time Profile (TTP) Query. Given a start node $s \in V$, a destination node $t \in V$, and a departure interval $I \subseteq \mathbb{R}$ we want to compute a *travel time profile*; that is, a function $f : I \to \mathbb{R}_{\geq 0}$ that maps every departure time $\tau \in I$ to the corresponding minimal possible time duration for traveling from s to t in G.

Minimizing Time-Dependent Travel Times with Additional Costs. Second, we consider the more general case, that the travel cost of a route can be different from the time needed to travel this route. As said before, the weight of an edge

$e \in E$ is a function $C_e : \mathbb{R} \to \mathbb{R}_{\geq 0}$ where $C_e(\tau)$ is the cost of traveling along the edge e when departing at time τ. We call C_e the *travel cost function (TCF)* of e. Without travel time information, however, we do not know where in time we are when traveling along the edge e. So, the TTF f_e is still necessary. As a consequence, the weight of e actually consists of both functions, f_e and C_e, but C_e carries the actual cost information.

In this thesis we only consider a pretty restricted subset of possible TCFs; namely, functions C_e of the form

$$C_e(\tau) = f_e(\tau) + c_e$$

with $c_e \in \mathbb{R}_{\geq 0}$ for all $e \in E$. We speak of time-dependent route planning *with additional time-invariant costs* in this case. This model seems to be simple. We will see, however, that user queries are already NP-hard in this setting (see Section 6.1). Again, we have two kinds of user queries depending on the fact whether the user gives a single departure time or a departure interval: *minimum cost queries* and *cost profile queries*.

Minimum Cost (MC) query. Given a start node $s \in V$, a destination $t \in V$, and a departure time $\tau_0 \in \mathbb{R}$ we want to compute the minimal possible travel cost in G as well as a corresponding path in G.

Cost Profile (CP) Query. Given a start node $s \in V$, a destination $t \in V$, and a departure interval $I \subseteq \mathbb{R}$ we want to compute a *cost profile*; that is, a function $D : I \to \mathbb{R}_{\geq 0}$ which maps every departure time $\tau \in I$ to the corresponding minimal possible cost for traveling from s to t in G.

Difficulty of Different Time-Dependent Queries. It must be noted that profile queries are usually much harder to answer than queries with a single departure time. And generalized time-dependent travel costs—compared to the case where travel costs and travel times are the same—also make everything much harder. In Table 1.1, as a consequence, difficulty increases from top left to bottom right: EA queries are the easiest kind of queries and CP queries are the hardest kind of queries. Note that the complexity of EA queries is due to the well-known fact that Dijkstra's algorithm can be adapted to work with time-dependent travel times as travel costs [35] (see Section 4.2.1) while assuming that road networks are sparse (i.e, each node has $\mathrm{O}(1)$ edges) and that all TTFs fulfill the FIFO property. That TTP queries need running time $\mathrm{O}(|V|^2 \log |V|) \cdot F \cdot |V|^{\mathrm{O}(\log |V|)}$ can be derived from Foschini et al. [36] (for a short summary see Section 1.3.2 on page 38). Note that the underlying road network is again assumed to be sparse. The NP-hardness of MC queries follows from the NP-hardness of a more special problem (see Ahuja et al. [3], for a short summary see page 39) as discussed in Section 6.1.1. CP queries are a generalization of MC queries and are as least as hard. Note that the complexities reported in Table 1.1 require that all TTFs in the underlying road network fulfill the FIFO property. Otherwise, querying would even be harder.

Periodic Time-Dependent Travel Costs. In this thesis all time-dependent travel costs are periodic. More precisely, all TTFs and all TCFs are periodic with a period of 24 hours. Regarding profile queries this enables us to choose $I := \mathbb{R}$ as interval of possible departure times. This means that profile queries compute the minimum time-dependent travel times and the minimum time-dependent travel costs for *all* departure times respectively.

Note that non-periodic TTFs and TCFs would not make a real difference for the techniques discussed in this thesis. In context of real-live applications an appropriate time window with a sufficient large period should be chosen. This may change the memory usage of course. Regarding the preprocessing time we expect a change by a constant factor.

1.1.3 Exact versus Inexact Route Planning

In practice, the accuracy of travel cost data delivered with road network data may be arguable. Not only because of potential weaknesses in the process of data collection but also because of the constantly changing conditions in the real world. A route computed by a route planning algorithm may be the optimal route with respect to the available travel cost data, but it may not be the best route for some specific traveler on some specific day. There may be unforeseen bad weather or unusual high traffic volume on that very day. The computed route may still be a good route, but not the best one. So, is the ability to compute *optimal* routes at all interesting? We answer this question with a clear yes.

To understand that, think of the possible alternatives. On the one hand, there are heuristic algorithms. Designing such algorithms may sometimes be easier than exact ones, but the routes computed by such algorithms can get arbitrarily bad. Even if bad routes are rare, a few dissatisfied users may already be enough to raise bad publicity. This, of course, can negatively affect the success of product or service. On the other hand, there are approximation algorithms with known error bound. Such algorithms would be a good alternative if the error is small enough. In the context of time-dependent route planning, however, developing such algorithms is still challenging. This is because of the usually non-linear structure of the time-dependent travel costs. Especially, this makes it hard to analyze the error of approximate algorithms. Here it must be noted that at least for EA and TTP queries with *continuous* TTFs, an *upper bound* of the maximum possible relative error can be stated depending on the maximum slope of the present TTFs [42]. For non-continuous TTFs and for generalized time-dependent travel costs this may be a different story.

So, we see that route planning algorithms should not only be fast. Route planning algorithms should also be *exact* in the sense, that the computed routes are *provably optimal*. Depending on the structure of the underlying travel cost model and depending on the type of user queries, however, fast and exact algorithms can be very difficult to design. So, inexact (or heuristic) algorithms can still be an alternative; namely,

- if we fail to design a sufficiently efficient exact algorithm, or
- if the speedup gained or the memory saved by the inexact algorithm justifies the loss in accuracy.

In any case, the inexact algorithm should be evaluated experimentally to ensure that the obtained routes are sufficiently well-behaved.

1.1.4 Contraction Hierarchies (CHs)

The last decades research resulted in numerous fast and exact route planning algorithms for road network with constant travel costs. *Contraction hierarchies (CHs)* are a very efficient and influential technique in this area [44]. The aim of this thesis is to adapt CHs to the needs of time-dependent route planning.

Like most other route planning techniques CHs work in two stages. In the first stage—the expensive *preprocessing*, which is usually performed off-line—we construct a special representation of the road network. In the second stage—the *querying*—we use this precomputed representation for fast and exact answering of one-to-one queries with very high throughput. The idea behind CHs is to exploit the inherent hierarchical structure of road networks. Some parts of a road network are usually more important than others in the sense that the majority of convenient connections leads through them. Accordingly, a CH is a *hierarchical* representation of the original road network where more important nodes are higher up in the hierarchy. Note that the term CH is used for both the speedup technique and the hierarchical representation of the road network.

The crucial property of a CH is that there is always an optimal path $\langle s \to \cdots \to x \to \cdots \to t\rangle$, such that $\langle s \to \cdots \to x\rangle$ only goes *upward* and $\langle x \to \cdots \to t\rangle$ only goes *downward* in the hierarchy. The guaranteed existence of such *optimal up-down-paths* in CHs enables the following very efficient query procedure: For given start node s and destination node t we perform a *bidirectional Dijkstra* search in the CH that only goes *upward*—a *forward* search starting from s and a *backward* search starting from t. Searching upward means that we only relax edges that lead higher up in the hierarchy. The efficiency of this approach is due to the fact that well-constructed CHs are usually *flat* and *sparse*. So, upward searches reach the top of a CH quite soon and the branching factor of the searches is not too large.

It is important to know that the guaranteed existence of optimal up-down-paths requires the presence of *shortcut* edges in the CH. A shortcut (edge) is an "artificial" edge that represents a path in the original road network. The CH consists of the *original edges* from the original road network together with several shortcuts and some additional information stored with every shortcut (for details see Section 2.3.1). Using the additional information, shortcuts (and hence up-down-paths containing shortcuts) can be *expanded* to the *original paths* they represent.

1.2 Contributions

This thesis describes how contraction hierarchies (CHs) [44]—the algorithmic framework for fast and exact route planning with non-negative constant travel costs recapitulated in Section 1.1.4—can be adapted to the needs of time-dependent route planning. The underlying goal is to enable fast and exact time-dependent one-to-one queries. Regarding EA and TTP queries, we succeed. Regarding MC queries, we only provide fast but inexact computation so far—but with negligible error in practice. CP queries have not been worked out with CHs yet.

The *time-dependent contraction hierarchies (TCHs)* described this thesis including its variants are far from being a straightforward extensions of the original CHs. Instead, several sophisticated algorithmic ingredients are necessary to make them run efficiently. This applies to both stages of TCHs, preprocessing as well as querying. The most important idea utilized in this thesis is the careful use of approximate computations to find small subgraphs that contain all relevant edges. The exact computations are performed afterwards, only on that subgraphs. This simple but effective idea enables us

- to perform bidirectional search in the context of time-dependent travel costs (this is important, as it is bidirectional search that makes CHs so effective),
- to considerably accelerate some very expensive exact computations without losing exactness, and
- to perform exact computations in the presence of space-saving approximate data which decreases the memory consumption of TCHs a lot.

1.2.1 Time-Dependent Travel Times with Additional Costs

Our first contribution is a thorough discussion of time-dependent travel costs and time-dependent minimum cost paths, as far as they are relevant to this thesis (see Chapter 3). For the case that travel costs are time-dependent travel times, this is not new. However, we provide a relatively detailed description, how the basic operations on TTFs can be realized efficiently (see Section 3.1.4). This might be interesting for people willing to put efficient time-dependent route planning into practice.

Regarding more general time-dependent travel costs, we consider the special case of time-dependent travel times (i.e., TTFs) with additional time-invariant (i.e., constant) non-negative costs (see Section 3.2). This not only includes a discussion of the corresponding class of time-dependent minimum cost (MC) paths (see Section 3.2.1) but also of the basic operations that are necessary to deal with this kind of time-dependent costs. This results in the definition of an algebraic structure that is not only powerful enough to express everything we need, but that is also well-behaved in the sense that it is closed under linking and merging (see Section 3.2.2). Algorithms for the basic operations on this algebraic structure are also discussed (see Section 3.2.3). Note that this algebraic structure is able to handle more general time-dependent shortest path problems; namely, the case that TTFs have points of discontinuity and that additional costs are time-dependent (i.e., piecewise constant). We provide a short discussion of the corresponding class of MC paths (see Section 3.2.4). This is necessary because the TCH-based MC queries described in this thesis have to deal with this kind of costs, even if the underlying original road network comes with time-invariant additional costs.

1.2.2 Algorithmic Ingredients

Our second contribution is a relatively detailed discussion of several basic time-dependent Dijkstra-like shortest path algorithms that we use as ingredients of TCHs (see Chapter 4). This includes a discussion of the structures that are common to all these algorithms (see Section 4.1). The Dijkstra-like algorithms are either *single label* searches running in forward direction (see Section 4.2), single label searches running in backward direction (see Section 4.3), or *multi-label* searches (see Section 4.4).

Some of the Dijkstra-like algorithms compute exact results, the others are approximate versions of the exact ones. The approximate versions play an important role in this thesis, because their exact counterparts are often very slow and the approximate versions run considerably faster. This is in fact the reason, why running approximate computations before exact ones can be used to speed up exact computations a lot. The faster approximate algorithms are used to reduce the search space. The slow exact algorithms are applied afterwards to the reduced search space to compute the desired exact result.

We provide proofs of correctness for all Dijkstra-like algorithms used in this thesis. All these proofs follow a similar pattern and some Dijkstra-like algorithms are even special cases of others. So, details of proofs can often be omitted.

1.2.3 Minimizing Time-Dependent Travel Times

Our third contribution is to make CHs [44] run efficiently when the time-dependent travel costs are the travel times (see Chapter 5). This includes several aspects that require careful consideration: preprocessing (see Section 5.2), fast exact EA and TTP queries (see Section 5.3), as well as space efficiency (see Section 5.4). Interestingly, the more space efficient setup enables us to further speed up exact TTP queries. Also, we consider heuristic inexact EA and TTP queries (see Section 5.5).

Preprocessing. For the fast and exact answering of EA queries and TTP queries we have to build a TCH structure first. This computationally expensive process is performed in a preprocessing step, just like in case of original CHs. It must be noted, however, that the construction of a TCH is much more expensive than the construction of a CH. In fact, we have to apply several ideas to make the TCH preprocessing reasonably fast. This includes

- cheap approximate time-dependent searches to quickly find a result or at least to derive small corridors in which expensive exact searches are done,
- caching of previously computed intermediate results, and
- thinning of the aforementioned corridors in a heuristic way.

This yields an exact TCH even if heuristic is used for intermediate steps. Also, the preprocessing parallelizes quite well for shared memory (see Section 5.2).

Note that the shared memory parallelization has been contributed by Vetter in his student research project [82]. Vetter also introduced the mentioned heuristic thinning of corridors and the caching of intermediate results when he was working on the implementation of the preprocessing as a student assistant (see Section 5.7 for details). His contributions are crucial for the preprocessing as they reduce its running time a lot.

Exact EA Queries. Fast and exact answering of EA queries with TCHs includes bidirectional search. This raises the problem that we would already have to know the earliest possible arrival time—but this is part of we want to compute. To overcome this, we perform an approximate time-dependent shortest path search as backward search. Afterwards, we perform a time-dependent variant of Dijkstra's algorithm but only using edges that have been touched by the backward search—the *downward* search as we call it. This yields the earliest possible arrival time for the given departure time as well as a corresponding optimal up-down-path. The up-down-path can be expanded to the corresponding optimal path in the original road network. Note that the result of the computation is exact even if the backward search is approximate (see Section 5.3.1).

Exact TTP Queries. Fast and exact answering of answering TTP queries with TCHs also includes bidirectional search. But in contrast to exact EA queries it is not a problem to apply bidirectional search directly. This is because of the fact that we want to compute the minimum travel time for *all* possible departure times and there is no dedicated arrival time hence. The resulting algorithm is simple and has feasible running time. But we can be much faster: To reduce the search space we perform an approximate bidirectional search first. The much more expensive exact bidirectional search is performed only on a subset of the edges touched by the approximate bidirectional search. This brings a considerable speedup while the result of the computation stays exact (see Section 5.3.2).

Saving Memory and Speeding up TTP Queries. TCHs—just like the original CHs—make intensive use of shortcut edges. In case of TCHs, however, this requires much more memory. The reason is that the time-dependent travel time information associated with the shortcuts is usually quite complex. In fact, it contains a lot of redundancy. This can be utilized to greatly reduce the memory consumption of TCHs: For shortcuts we only store *approximated* travel time information and for original edges we store *exact* travel time information. The resulting structure—we call it an *approximated TCH (ATCH)*—needs much less space than the underlying TCH (see Section 5.4.1).

Although ATCHs contain a lot approximate data, they can be used to answer exact EA and TTP queries (see Sections 5.4.2 and 5.4.3). The resulting algorithm for EA queries is still very fast, although it runs moderately slower than the query algorithm we use with TCHs. Our method performs approximate searches on the ATCH to obtain small subgraphs that usually contain shortcuts. Expanding all these shortcuts we get a subgraph of the original road network. There, all stored travel time information is exact and we apply exact time-dependent searches. In case of TTP queries we even get a considerable speedup. We achieve this by contracting *whole subgraphs*.

Inexact EA and TTP Queries. In Section 1.1.3 we argued that inexact heuristic route planning is still interesting when the gained speedup justifies the loss in accuracy. For inexact TTP queries (see Section 5.5.2) this is definitely the case as we save a lot of running time by accepting a small error. We achieve this by using an *inexact TCH* instead of an exact one. There, we not only store the travel time information of the shortcut edges but also of the original edges in an approximate manner (see Section 5.5.1). Inexact TCHs are similarly space efficient as ATCHs. It must be noted that inexact TCHs can also be used to perform inexact EA queries (see Section 5.5.3). Though ATCHs already provide fast, exact, and space efficient EA queries, an extra ATCH may be too memory-consuming if we already have an inexact TCH in memory.

Experimental Evaluation. To support our claims we have implemented all the techniques mentioned above in C++ and report an extensive experimental evaluation. As inputs we use the road networks of Germany and Western Europe with time-dependent travel time information. Both networks have been provided by PTV AG [71] (see Section 5.6).

1.2.4 Additional Time-Invariant Costs

Our fourth contribution is to make CHs [44] run efficiently when the time-dependent travel costs are time-dependent travel times with additional time-invariant costs (see Chapter 6). It turns out that the answering of MC queries and CP queries is much more complex than answering EA queries and TTP queries, even in this relatively restricted special case. So, we only come up with a *heuristic* preprocessing procedure. The resulting heuristic generalized TCHs can be used to answer MC queries fast. The resulting routes are not necessarily optimal, which means that exactness is lost. But note that we only consider small errors in our experiments

Complexity. It turns out that the size[2] of a cost profile (CP) can be exponential in the number of nodes of the road network . This corresponds to the fact that one-to-one MC queries are NP-hard, even in the relatively special case that the travel costs are time-dependent travel times with additional time-invariant costs. This follows from the NP-hardness of an even more restricted special case considered by Ahuja et al. [3], where the choice of additional time-invariant costs is very limited (we recapitulate this in Section 1.3.2 on page 39). We report a translation of their proof into the context of additional time-invariant costs that can be chosen freely (see Section 6.1).

Heuristic Preprocessing. After discussing what makes node contraction problematic in the presence of additional time-invariant costs, we describe our heuristic preprocessing procedure. To obtain a heuristic TCH from a given time-dependent road network with additional time-invariant costs, we use essentially the same techniques as without additional time-invariant costs. Some modifications have to be made of course. The resulting heuristic TCH structure still guarantees

[2]In this work we model travel time functions (TTFs) as continuous piecewise linear functions (see Section 1.1.2). As a consequence, travel time profiles (TTPs) and cost profiles (CPs) are also piecewise linear functions. The *size* of a TTP or of a CP, respectively, is simply the number of its bend points.

that an up-down-path from a start to a destination node is present. However, this up-down-path is not necessarily an optimal one (see Section 6.2).

Heuristic Minimum Cost Queries. The algorithm we use for fast MC queries is a modification of TCH-based EA query. The main difference is that forward and downward search are performed in the manner of a multi-label search. The resulting method is exact if this is the case for the underlying generalized TCH structure. There, a TCH structure counts as exact if the existence of *Pareto prefix-optimal* MC up-down-paths is guaranteed for every possible combination of start node, destination node and departure time. Unfortunately, our preprocessing procedure is not able to provide such exact TCHs. However, according to our experiments, the query algorithm runs fast on our heuristic TCH structures and the returned routes are only slightly worse than the optimal ones (see Section 6.3).

Experimental Evaluation. We have implemented our heuristic preprocessing procedure and our MC query procedure in C++. Our experimental evaluation again uses the road network of Germany provided by PTV AG [71]. The additional time-invariant costs are proportional to travel distance to obtain a very simple approximation of energy costs, additionally combined with motorway tolls. Regarding the heuristic preprocessing, we evaluate the parallel running time of node ordering—though it must be noted that our node ordering procedure is implemented only as a prototype and probably slower than necessary. We also evaluate the running time of MC queries. Especially, we evaluate the relative error of MC queries that turns out to be negligibly small.

1.3 Related Work

There has been a lot of research on route planning in road networks in the last several years. This section gives a brief overview of the results related to this thesis. Overview papers have been provided by Delling et al. [25], by Delling and Wagner [28], by Wagner [83], by Bast et al. [4], and by Sommer [80]. Much research on route planning deals with constant travel costs (see Section 1.3.1). More recently, time-dependent route planning also gained considerable interest (see Section 1.3.2). Other kinds of generalized travel costs increasingly get into focus (see Section 1.3.3). If not stated otherwise, all route planning techniques discussed in this section are exact.

Many of the algorithmic methods for route planning—often referred to as *speedup techniques*—are either *hierarchical* techniques, *goal-directed* techniques, or *combinations* of these. Speedup techniques usually work in two stages: *preprocessing* and *querying*. In the preprocessing stage, which is usually expensive and thus performed off-line, we construct a special representation of the road network that is especially well suited for fast and exact answering of route planning queries. The properties of this representation depend on the particular speedup technique. Having finished the preprocessing, we switch over to the querying stage and use the resulting representation for fast and exact answering of one-to-one queries for example. This principle has already been described in the context of CHs, which are one of the hierarchical speedup techniques (see Section 1.1.4).

1.3.1 Route Planning with Constant Travel Costs

The basic method for computing optimal paths in road networks with constant travel costs is Dijkstra's well known algorithm [33] (more details can be found in Section 2.2.3). For general graphs with non-negative constant edge weights no faster algorithm has been developed since. For the special case that the graph represents a road network, much faster algorithms have been found; namely, the speedup techniques mentioned above. On road networks, these algorithms run several orders of magnitudes faster than Dijkstra's algorithm. This may, in fact, be the reason where the term "speedup technique" comes from.

Usually, the performance of a speedup technique is compared to the performance of Dijkstra's algorithm by an experimental evaluation. The resulting figures are often reported in terms of

- *speedup*, which is the average running time needed by Dijkstra's algorithm divided by the average running time that the speedup technique needs to answer user queries,
- *preprocessing time*, which is the time the preprocessing stage needs to generate the representation of the road network that the speedup technique uses during the query stage, and
- *space overhead*, which is how much this representation increases the memory usage compared to the graph representation used by Dijkstra's algorithm.

Table 1.2 reports speedup, preprocessing time, and space overhead of all the speedup techniques for answering exact one-to-one queries with constant travel costs that are described in this section; excluding mobile, approximate, and customizable CHs.

Speedup and space overhead are reasonably machine-independent measures. By now, reporting the speedup has even become standard in research on route planning. To obtain a reasonably machine-independent measure of preprocessing time, we divide the measured absolute preprocessing time by the average running time of Dijkstra's algorithm measured on the same machine. This yields the *relative preprocessing time* reported in Table 1.2.

			space	preprocessing time		query time	
	data		overh.	absolute	relative	avg.	speed
method	from	machine	[B/n]	[h:m]	[× Dijk.]	[ms]	up
			Western Europe, PTV				
Dijkstra	[78]	Opteron 2.0 GHz[a]	0	0:00	*0*	*6 060.4*	*1*
HHs	[78]		48	0:13	*129*	0.61	9 935
HNR	[78]		2.4	0:15	*149*	0.85	*7 130*
Dijkstra	[11]	Opteron 2.6 GHz[b]	0	0:00	*0*	5 591.6	*1*
arc-flags	[77]		81	196:29	*126 500*	0.80	*6 990*
ALT	[26]		512	1:08	*730*	19.6	*285*
CALT	[11]		15.4	0:11	*118*	1.34	*4 173*
CHs	[11]		−2.7	0:25	*268*	0.18	*31 064*
CHASE	[11]		12	1:39	*1 062*	0.017	*328 918*
SHARC	[10]		14.5	1:21	*869*	0.29	*19 281*
HL	[1, 11]	Xeon 3.33 GHz[c]	≈ *313*	3:16[d]	n/a	0.0005	*5.5 mio*

Table 1.2. Comparing different speedup techniques ("method") for one-to-one queries with constant travel costs with respect to space overhead ("overh.") in byte per node ("B/n"), preprocessing time, and query time. Preprocessing time is not only reported as absolute values in hours and minutes but also relative to the average ("avg.") running time of one-to-one Dijkstra search on the same machine. Query time is reported in terms of the average measured running time as well as the speedup compared to the average running time of one-to-one Dijkstra search, also on the same machine.
All results are measured using instances of the Western European road network provided by PTV AG [71] for scientific use, all with about 18 mio nodes. The number of edges varies a little more: 42.2 mio [78], 42.6 mio [10, 11, 26], and 44 mio [77, 1]. Figures printed in italic are not directly present in the underlying data but have been calculated. The table is partitioned with respect to the machine on which the experiments are performed. In case of Opteron 2.0 GHz, the running time of one-to-one Dijkstra is calculated from speedup and query time of HHs. In case of arc-flags, relative preprocessing time and speedup are reported compared to a one-to-one Dijkstra that runs on the same machine but with SUSE Linux 10.1 instead of 10.3. In case of HL, space overhead and speedup are calculated from the figures in [1] and [11]. In case of speedup this utilizes that [1] reports a scale factor for running times from [11]. The reported space overhead of HL is only an estimate assuming exactly 18 mio nodes.

[a] AMD Opteron 270 with 2.0 GHz, 8 GB RAM, 2 × 1 MB of L2 cache, and SUSE Linux 10.0.

[b] AMD Opteron 2218 with 2.6 GHz, 16 GB RAM, 2 × 1 MB of L2 cache, and SUSE Linux 10.3 (except for arc-flags where SUSE Linux 10.1 is used)

[c] 2 × Intel Xeon with 3.33 GHz and 6 cores each, 96 GB RAM; 6 × 64 kB L1, 6 × 256 kB L2, and 12 MB L3 cache per CPU; Windows 2008R2 Server.

[d] In parallel with 12 threads.

Hierarchical Techniques. Hierarchical speedup techniques exploit the inherent hierarchical structure of road networks to provide fast and exact answering of route planning queries. Some parts of a road network are usually more important than others in the sense that optimal routes rather lead through them than through other parts. This is reflected by the structure of the special representations of road networks that are generated by the preprocessing stages of hierarchical speedup techniques. These preprocessed representations are usually hierarchical in the sense that the more important parts of a road network are located on a higher *level* than the less important parts. What makes hierarchical speedup techniques fast, is the property that, in hierarchical representations, shortest paths can be found using bidirectional Dijkstra searches that only go *upward*. Usually, this needs little running time, because hierarchical representations are usually flat and sparse. This fundamental idea has already been mentioned in the context of CHs (see Section 1.1.4).

Highway hierarchies (HHs) [75]. HHs are one of the earlier hierarchical speedup techniques for fast and exact route planning. Compared to CHs they are more complicated, especially with respect to preprocessing but also with respect to querying. Similar to CHs, the query algorithm of HHs is a bidirectional Dijkstra search that goes upward. But before going further upward, HHs search the neighborhood of the *entrance node* to the current hierarchy level. What the neighborhood of a node is, has to be determined by the preprocessing and to be maintained during querying. Note that HHs apply shortcuts to *bypass* several nodes. During the query this brings a further speedup.

Highway Node Routing (HNR) [74, 78]. HNR has a simpler query procedure than HHs. HNR, in contrast to HHs, is not aware of the neighborhood that a node has on a certain level. This requires the presence of shortcut edges to ensure correctness; that is, to ensure that optimal routes are computed (in contrast to HHs where shortcuts are not used to ensure correctness but only to speed up the computation). The shortcut edges are added to the hierarchy during preprocessing. There, sets of *covering paths* are computed; that is, sets of paths that are part of optimal routes. The shortcuts added to the hierarchy represent these paths. This is necessary because the paths represented by the shortcuts contain downward edges which can not be found by a query algorithm that only searches upward.

Contraction Hierarchies (CHs) [44]. CHs are an extreme case of HNR. Conceptually, CHs and HNR have the same query algorithm, but CHs have a simpler preprocessing procedure. Every node of a CH structure has a level on its own. As a consequence, shortcuts edges are restricted to represent paths of two hops only. This simplifies the decision whether a shortcut is necessary: A shortcut $u \to v$

representing a path $\langle u \to x \to v\rangle$ is added to the hierarchy only if $\langle u \to x \to v\rangle$ is a shortest path. To decide this, a one-to-one Dijkstra search from u to v can be used (see Section 2.3 for details).

Mobile CHs [76]. CHs have also been adapted to the needs of *mobile* devices. Usually, mobile devices have little main memory. So, the CH structure has to be stored on flash memory and the primary bottleneck is the number of blocks read from flash memory during querying. To remedy this, the adjacency array that represents the CH structure is reordered in a way that the number of blocks read is reduced. To provide fast generation of driving directions, shortcuts have to be expanded fast. To achieve that, the original path represented by each shortcut is computed during preprocessing and stored in a table on flash memory.

Approximate CHs [45]. This inexact version of CHs computes routes that are not too far away from the optimal ones. The idea is that less shortcut edges can be added during preprocessing to reduce preprocessing time, query time, and memory consumption. But on road networks with travel time as travel cost, this approach only brings partial improvements. However, if the travel cost is the distance or if the underlying network is a grid or a sensor network, then the preprocessing time, the query time, and the memory consumption are significantly smaller.

CHs with Flexible Edge Restrictions [41]. This variant of CHs computes optimal routes in the presence of *dynamic* edge constraints. The preprocessing must hence provide a CH structure being able to deal with all possible configurations of constraints. To achieve this, more shortcut edges must be added to the CH structure than without constraints, and even parallel shortcuts are used. The resulting running times for preprocessing and querying are significantly larger than without edge constraints but definitely good enough for practice. Combining with Landmark A* (ALT [47], see page 34) either reduces the time needed for querying or for preprocessing.

Many-to-Many Queries. Hierarchical techniques are not only suited for one-to-one queries but also for *many-to-many* queries. There, we do not have single start and destination nodes but a start set $S \subseteq V$ and a destination set $T \subseteq V$. The result of the computation is an $|S| \times |T|$-table containing the exact shortest path distances from all nodes in S to all nodes in T. The idea is essentially to build $|S| + |T|$ upward search spaces and then to intersect these search spaces in an efficient way to compute the table. This has first been done using HHs [56, 78] and then using CHs [43, 44], where it runs even faster. The resulting running time needed to compute a complete $|S| \times |T|$-table lies far below the running time needed by $|S| \cdot |T|$

one-to-one queries. Using CHs with $|S| = |T| = 10\,000$, the table can be computed in 10.2 sec. Compared to $10\,000 \cdot 10\,000$ one-to-one queries with CHs the speedup is 1 490. Compared to 10 000 average one-to-one Dijkstra searches (which may be a reasonable approximation of 10 000 one-to-many Dijkstra searches) this is a speedup of 5 481.

PHAST [22]. The PHAST technique is an adaption of CHs for the fast and exact answering of *one-to-all* queries: Given a start node s we want to compute the shortest path distance from s to all other nodes in the road network. The basic idea is to build the complete upward search space of s in a CH structure and then to search downward from all nodes that lie in that upward search space. There, the downward search is not a Dijkstra search, but exploits the fact that the downward edges of a CH structure form a DAG. So, processing all downward edges in a convenient order turns the downward search into a linear sweep with high locality which is very cache efficient. This results in a speedup of 16.4 compared to a one-to-all Dijkstra search.

Answering a *several-to-all* query in this setup—that is, computing the shortest path distances from k start nodes to all other nodes—leads to an improved speedup compared to k one-to-all Dijkstra searches: a speedup of 76.2 or 29.2 with or without SSE instructions, respectively, for $k = 16$. Without SSE instructions, PHAST parallelize well on shared memory architectures. With SSE instructions, however, the speedup obtained with 4 cores is only around 2 (though the absolute running times are still better with SSE). There is also a GPU implementation of PHAST: Answering several-to-many queries with an Nvidia GTX580 clocked at 772 MHz brings a speedup of 1 279 for $k = 16$.

Hub labels (HL) [1]. The idea of hub labels is to query shortest distances in a road network without actually searching in the road network. A *forward label* $L_f(u) \subseteq V \times \mathbb{R}_{\geq 0}$ and a *reverse label* $L_r(u) \subseteq V \times \mathbb{R}_{\geq 0}$ is attached to every node $u \in V$. For every two nodes $s, t \in V$ the labels $L_f(s)$ and $L_r(t)$ both contain a node x that lies on a shortest path from s to t; that is, $(x, c_f) \in L_f(s)$ and $(x, c_r) \in L_r(t)$, such that a shortest path $\langle s \rightarrow \cdots \rightarrow x \rightarrow \cdots \rightarrow t \rangle \subseteq G$ exists and $c_f + c_r$ equals the shortest path distance from s to t. If one represents all node labels as sequences that are sorted with respect to the node id, shortest distance queries can simply be answered by scanning through the labels $L_f(s)$ and $L_r(t)$ together.

The labels are obtained from CHs [44] by considering the forward and backward search space of every node; that is, all nodes that are reachable in upward direction by following forward and backward edges respectively. Applying the pruning technique stall-on-demand [44, 74] offline (i.e., not in the query stage but during preprocessing) as well as other pruning and compression techniques reduces the size of the resulting labels a lot. This leads to the very small average query time of 0.5 μsec on continental size road networks, but preprocessing takes more than three hours, with 12 parallel threads running in shared memory.

Customizable Route Planning. Customizable route planning introduced by Delling et al. [23] attends to the problem that small query times usually come at the cost of large preprocessing times. This makes it difficult to take the current traffic situation into account. To overcome this, Delling et al. split the preprocessing into two sub-phases: a first sub-phase that is independent from the travel costs and solely depends on the topology of the road network, and a second sub-phase that computes the information depending on the travel costs. It turns out that the second sub-phase, the *customization*, can be performed quite fast; namely, within 0.35 sec for the Western European road network using 12 cores in shared memory [29]. So, the preprocessed structures can be updated easily to provide route planning depending on the latest traffic data. The query times stay below 2 msec. The first sub-phase of preprocessing can be done within 13 minutes.

Very recently, Dibbelt et al. described a customizable version of CHs [31]. The first sub-phase of the preprocessing computes a *metric-independent CH*; that is, a CH where no shortcut edge is omitted. This means the contraction of a node produces a clique in the remaining graph. It turns out that the original node order procedure of CHs [44] is not feasible with metric-independent CHs, as it provokes too many shortcut edges. Instead, a different node order is build from top to bottom by recursively dividing the road network into partitions with possibly small separators [51]. The nodes in the current separator are attached to the current lower end of the node order. For the Western European road network, this takes 69 hours sequential running time (see the extended version [32] of the article [31] by Dibbelt et al.). All shortcuts of the corresponding metric-independent CH can be determined within 15.5 sec. The customization, which takes 0.74 sec using 16 cores, assigns travel costs to the original edges and propagates them upward to the top of the CH. The query algorithm processes the nodes in appropriate order without using a priority queue. It answers one-to-one queries within 0.41 msec.

It must be noted that the reported figures include support for turn costs in case of CRP, but no support for turn costs in case of customizable CHs.

Goal-Directed Techniques. The idea behind goal-directed speedup techniques is to perform a biased Dijkstra search from the start to the destination node. That is, a one-to-one Dijkstra search that does not spread out in a more or less circular way—as the original Dijkstra search does—but rather in direction of the destination node. Effective goal-directed techniques utilize information computed in a preprocessing stage.

A Search.* A typical example of a goal-directed technique is A* search. Though this well-known algorithm is originally described by Hart et al. [49], we rather follow the style it is presented in the textbook by Mehlhorn and Sanders [61] (see Section 2.2.5 for details). A* search estimates the shortest path distance $\mu(u,t)$ from each processed node u to the destination node t by a *lower bound* $\pi_t(u) \leq \mu(u,t)$, and this estimate is used to change the order in which Dijkstra's algorithm processes the nodes (π_t is also known as *potential function*). If we manage to choose a relatively tight lower bond, then this order is changed in a way that only few nodes are processed before the destination node is reached. The running time can be reduced significantly this way. All known lower bounds that are sufficiently tight on road networks, however, require precomputed information.

Landmark A (ALT)* [47]. An example of A* search with a sufficiently tight lower bound is based on *landmarks* and the *triangle inequality*. This setup is known as *landmark A** or *ALT*.[3] A set of landmarks is a small set $\{\ell_1,\dots,\ell_k\} \subseteq V$ of wisely chosen nodes. Together with every node $u \in V$ we store the $2k$ shortest path distance values $\mu(u,\ell_i),\mu(\ell_i,u)$ with $1 \leq i \leq k$. The k landmarks and the $2k \cdot |V|$ shortest path distance values are computed during preprocessing. In the query stage, we use

$$\pi_t(u) \;:=\; \max\left\{\mu(\ell_i,t)-\mu(\ell_i,u),\; \mu(u,\ell_i)-\mu(t,\ell_i) \;\middle|\; 1 \leq i \leq k\right\}$$

as lower bound of $\mu(u,t)$ whenever a node u is processed. Experiments show that this lower bound is tight enough to achieve a significant speedup over the original Dijkstra search, which does not have a bias towards the destination node. ALT is usually performed in a bidirectional manner, though bidirectional ALT requires some sophistication.

Arc-Flags [57, 63]. Goal-directed techniques are not necessary variants of A* search. An example is the *arc-flags* technique. The idea is to partition the road network into k regions $R_1,\dots,R_k$ and then to annotate every edge $e \in E$ with k so called *arc-flags* $a_1^e,\dots,a_k^e \in \{\mathit{true}, \mathit{false}\}$. The i-th arc-flag a_i^e of an edge e is set to *true* if, and only if, e lies on a shortest path that leads into the region R_i. Whether this is the case or not, has to be found out in a pretty expensive preprocessing step: For every region R_i and each of its *border nodes* we perform a backward Dijkstra search that starts at the respective border node. Then we iterate over all edges e of the resulting shortest path tree setting $a_i^e := \mathit{true}$ each. The partitioning of the road network is also part of the preprocessing. Note that the choice of the regions during partitioning influences the running times, especially of preprocessing.

[3] In fact, the three letters ALT stand for A*, landmarks, and triangle inequality.

The query algorithm of the arc-flags technique is a very simple modification of Dijkstra's algorithm: Assume we want to answer a one-to-one query with a start node s and a destination node t where t lies in region R_j. Then, before relaxing an edge e we always check whether $a_j^e = true$ is fulfilled. If this is not the case, we do not relax e. Uniting regions to "super regions" on the next higher level and applying this idea recursively makes arc-flags a multilevel approach. Compared to simply increasing the number of regions this saves space and preprocessing time. The arc-flag technique becomes a goal-directed multi-level approach this way. A further improvement is to apply arc-flags to bidirectional search. This is nothing but a bidirectional Dijkstra where both forward and backward search are modified in the way just described. Of course, this requires that arc-flags are computed for both search directions. This increases preprocessing time and memory usage, but the query time is reduced a lot.

Combinations. Hierarchical and goal-directed speedup techniques can be considered as orthogonal approaches to some degree. This suggest that a combination of hierarchical and goal-directed techniques may yield even better speedups. Several results show that this is really the case. Here we only recapitulate *core-based routing* combined with ALT (Core-ALT or CALT), arc-flags combined with shortcuts (SHARC), and CHs combined with arc-flags (CHASE).

Combinations of goal-directed techniques with other speedup techniques seem also to be motivated by the fact that the preprocessing of goal-directed techniques (and especially of arc-flags) tends to be expensive. So, goal-directed techniques are often applied only to a smaller graph that is derived from the original road network. CALT, SHARC, and CHASE are examples of this approach.

Core-Based Routing with ALT (Core-ALT or CALT) [11]. Core-based routing means that a two-level hierarchy is created while shortcuts are added to the upper level to represent two-hop paths that contain downward edges. This is actually similar to CHs. CHs, however, have much more levels and the selection of levels is also different. The query algorithm is, as it is usual when hierarchies are used, a bidirectional Dijkstra search that goes upward. On the upper level (the *core*) an additional goal-directed speedup technique is used, in this case landmark A* (ALT). The resulting query is actually a bidirectional upward search in a two-level hierarchy where ALT is used to accelerate the searches on the upper level. CALT, compared to traditional ALT, shows better query times, better preprocessing times, and needs less memory.

Arc-Flags with Shortcuts (SHARC) [10]. SHARC improves the arc-flags technique, which we described before, by adding shortcuts. The idea is to remove the paths that are represented by the shortcuts from the graph during the arc-flags preprocessing. As a consequence, the preprocessing takes less time. Having computed the arc-flags, the removed paths are re-added to the road network and the arc-flags $a^e_1, \ldots, a^e_k$ of the first edge e of such a path are all set to *false*—except for flags a^e_i where the region R_i is crossed by the path. The arc-flags of all other edges of the path could be set to *true*. Better arc-flags, however, can be obtained by propagating the arc-flags information backward from the end of the path to its beginning. Applying this approach recursively makes SHARC a goal-directed multi-level approach, just like in case of arc-flags without shortcuts. The resulting query and preprocessing time as well as the memory usage are much better than for arc-flags without shortcuts.

CHs with Arc-Flags (CHASE) [11]. CHASE, like CALT, uses hierarchy to obtain the subgraph to that the goal-directed technique is applied. Here, the hierarchy is a CH structure and the goal-directed technique is the arc-flags technique. This upper part is called the *core*, just like in case of CALT. The query algorithm is in principle a CH query, but with an "interruption". This means the query has two phases: The first phase is the original CH query (i.e., a bidirectional upward search) with the modification that the search is *pruned* at core nodes. Also, core nodes are added to a target set S or a destination set T, respectively, if they are reached by the forward or backward search. The second phase is now able to determine the *target core regions*; that is, all core regions $R_{i_1}, \ldots, R_{i_\ell}$ with $R_{i_j} \cap T \neq \emptyset$. The *source core regions* are defined analogously as the core regions that intersect with S. Then, the second phase continues the bidirectional CH query but only relaxing edges where the arc-flags for the target and source regions are *true* respectively. There is also *partial* CHASE. In this case the core is not completely contracted or only partly contracted. This reduces preprocessing time but increases query time.

1.3.2 Time-Dependent Route Planning

Besides the work presented in this thesis there are also other results on route planning with time-dependent travel costs. Most of that work, however, deals with the special case that the time-dependent travel costs are travel times.

Fundamental Results. Before we focus on road networks, we first summarize some fundamental results that deal with general time-dependent graphs.

Time-Dependent Dijkstra [35]. An early result on route planning with time-dependent travel costs is a time-dependent variant of Dijkstra's algorithm described by Dreyfus. This *time-dependent Dijkstra* deals with the special case that the travel costs are travel times. It answers EA queries as described in Section 1.1.2. Dreyfus does not mention, however, that his algorithm requires that all TTFs fulfill the FIFO property. The algorithm works in principle the same way as the original Dijkstra's algorithm. Especially, nodes are never reinserted into the priority queue (for details see Section 4.2.1). The only difference is how new node labels are computed during edge relaxation. With the assumption that road networks are sparse graphs (i.e., every node has $\mathrm{O}(1)$ edges) this yields that time-dependent Dijkstra needs $\mathrm{O}(\log F \cdot |V| \log |V|)$ time, where F is the maximum size of any TTF in the road network. This is because TTFs can be evaluated in $\mathrm{O}(\log F)$ time if binary search is used to look up the required bend points of a TTF. In this thesis. binary search is *not* used for evaluating TTFs. Instead, we use a bucket data structure (see Section 3.1.4 on page 90) that enables us to evaluate TTFs in constant average time (under the assumption that the x-values of the bend points are uniformly distributed).

General Time-Dependent Travel Times [69]. Orda and Rom consider quite general time-dependent travel times where TTFs may violate the FIFO property or may even have points of discontinuity. Without FIFO property waiting can be beneficial and optimal paths may include intermediate stops hence. If waiting is allowed at *every* node, however, time-dependent Dijkstra can still be used to answer EA queries correctly. The authors achieve this by using the auxiliary TTF f'_e with $f'_e(\tau) = \min\{f_e(\tau + \tau') + \tau' \mid \tau' \geq 0\}$ instead of the original TTF f_e for every edge e (note that f_e fulfills the FIFO property). All waiting times can be derived easily after the time-dependent Dijkstra has terminated. Orda and Rom also describe how to answer TTP queries (see Section 1.1.2) in this setup. Their algorithm is similar to the travel time profile (TTP) search that we describe in Section 4.2.2 but without priority queue. Accordingly, their version of profile search works in a Bellmann-Ford-like[4] and not in a Dijkstra-like manner.

If waiting is forbidden completely, everything gets much harder: The authors show that no finite optimal path may exist or that optimal paths may include loops in this case. They also state that forbidden waiting makes the problem NP-hard (a proof can be found in a manuscript by the authors [68]). By allowing waiting at the start node and at every node having an incoming edge with a point of discontinuity, however, the authors are again able to apply time-dependent Dijkstra and their version of profile search—provided that all TTFs f_e with $e \in E$ are piecewise continuous. Again, the waiting times are computed in a subsequent step.

General Time-Dependent Travel Costs [70]. In another article, Orda and Rom consider time-dependent travel costs beyond travel times. The allowed TCFs even

[4]The Bellmann-Ford algorithm is a well-known method to compute shortest paths if travel costs are constant but can be *negative*. It is described in several textbooks including, for example, the one by Mehlhorn and Sanders [61].

include piecewise continuous functions. TTFs need not to fulfill the FIFO property, but they have to be continuous (such TTFs are in fact less general than the TTFs in the article discussed before [69], where time-dependent costs are travel times). Again, waiting on nodes can be beneficial. Each node has a *parking weight density* that specifies the times when waiting is allowed at this node and how much it costs for arbitrary time intervals. Interestingly, finite optimal paths may not exists even if all TTFs are constant and waiting is completely forbidden.

The authors describe an algorithm that answers minimum cost (MC) queries if a finite optimal path exists for the given start node, destination node, and departure time. Remarkably, this method is *not* a multi-label search, although it runs forward in a setup where prefix optimality in the classical sense is not guaranteed.[5] Instead, it works similar to travel time profile search (see Orda and Rom [69] as well as Section 4.2.2) and backward cost profile search (see Section 4.3.5), even though only a single departure time is given. The algorithm works in the manner of the Bellmann-Ford algorithm and computes node labels that are cost profiles mapping arrival time to travel cost. This works, because a relatively restricted (and also quite technical) notion of prefix-optimality is provided.

Time-Dependent Travel Times with FIFO Property. Dean provides a nice introduction to time-dependent travel times where the FIFO property is fulfilled [17]. He discusses different kinds of queries in this setup. These are one-to-one, one-to-all, all-to-one, and all-to-all queries, each for a single departure time or a departure interval as well as a single arrival time or an arrival interval. This way he directly obtains a backward version of time-dependent Dijkstra computing latest departure times for given destination node and arrival time.[6] Dean also describes a label setting algorithm to compute TTPs and conjectures the size of TTPs to be superpolynomial.

Foschini et al. [36] consider the complexity of TTP queries when the FIFO property is fulfilled. As their main result they show that the worst-case size of a TTP lies in $|V|^{\Omega(\log|V|)}$ and $K \cdot |V|^{O(\log|V|)}$, where K is the total number of bend points over all TTFs in G. This settles the conjecture made by Dean [17] just mentioned. For restricted TTFs where all line segments have slopes from a set $\{-1, \alpha^{-\beta}-1, \dots, \alpha^{-1}-1, 0, \alpha^{1}-1, \dots, \alpha^{\beta}-1\}$ with $\alpha > 1$ and $\beta \in \mathbb{N}$ (and

[5]We say that a shortest path problem exhibits *prefix optimality* if prefix paths of optimal paths are optimal paths themselves; or, in a weaker form, there always exists an optimal path with this property, even if some optimal paths may not fulfill it. Lack of prefix-optimality usually spoils the applicability of Dijkstra's algorithm and similar single-label searches. Switching to multi-label search [48, 60] (see Section 2.2.6 for a short introduction) can often fix this problem (though other solutions may be possible).

[6]The backward version of time-dependent Dijkstra works in terms of departure time function (see Section 3.1.1 on page 81). Latest departure interval search (see Section 4.3.4) is an approximate version of this backward time-dependent Dijkstra.

positive discontinuities[7] are also allowed), Foschini et al. determine polynomial worst-case size for a TTP. They also describe an algorithm for one-to-all TTP queries that has in principle a running time of $\mathrm{O}((\log|V|)^2 \cdot |E|)$ times the total size of the computed TTPs; or, more precisely, $\mathrm{O}((\log|V|)^2 \cdot |E|) \cdot K \cdot |V|^{\mathrm{O}(\log|V|)}$ time. For sparse graphs (i.e., every node has $\mathrm{O}(1)$ edges) this reduces to $\mathrm{O}(|V|\log|V|) \cdot K \cdot |V|^{\mathrm{O}(\log|V|)}$ time. With F being the maximum complexity of any TTF in the graph, we can write this as $\mathrm{O}(|V|^2 \log|V|) \cdot F \cdot |V|^{\mathrm{O}(\log|V|)}$, or even $F \cdot |V|^{\mathrm{O}(\log|V|)}$ with the lower order terms hidden in the asymptotically written exponent. Dehne et al. [18] describe an algorithm for one-to-one TTP queries with a running time of $\mathrm{O}((M+K)(|E|+|V|\log|V|))$, where M is the size of the computed TTP.

Doing $\mathrm{O}(K\varepsilon^{-1}\log R \cdot \log(L\varepsilon^{-1}))$ time-dependent Dijkstra searches in forward and backward direction, Foschini et al. manage to answer an approximate one-to-all TTP query with relative error ε; where R is the maximum occurring factorial elevation of a concave section of any TTP, and L is proportional to the maximum occurring spread with respect to the X-axis of such a section. The resulting TTPs have size $\mathrm{O}(K\varepsilon^{-1}\log R)$ each. Another interesting result is that the best EA path in a given departure time *interval* can be found in polynomial time. To do so, the authors perform several time-dependent Dijkstra searches in forward and backward direction. Dehne et al. also discuss an approximate method for one-to-one TTP queries. For given absolute error ε_{abs} and a departure interval of width Δ, they perform $\lfloor \Delta/\varepsilon_{\text{abs}} \rfloor$ time-dependent Dijkstra searches in backward direction.

Minima of TTFs as Additional Time-Invariant Costs [3]. Ahuja et al. examine discrete time-dependent travel times with quite restricted additional time-invariant costs that can also be *negative*. There, a TCF C_e of an edge e is of the form

$$C_e : \tau \mapsto \beta f_e(\tau) + (\alpha - \beta)\min f_e$$

where $f_e : \mathbb{Z} \to \mathbb{N}_{\geq 0}$ is the TTF of e and $\alpha, \beta \geq 0$. Note that α and β are chosen *globally* for the complete graph and thus equal for all edges. For $\alpha > \beta > 0$, we obtain a quite restricted special case of time-dependent travel times with additional time-invariant costs; namely, with TCFs of the form

$$C_e : \tau \mapsto f_e(\tau) + \frac{(\alpha - \beta)}{\beta}\min f_e \ ,$$

which is the same as $C_e : \tau \mapsto f_e(\tau) + \lambda \min f_e$ with *globally* chosen $\lambda > 0$. For $0 \leq \alpha < \beta$, we obtain a restricted version of time-dependent travel times with negative additional time-invariant costs, which is no longer a special case of the time-dependent minimum cost routing problem considered in this thesis. The

[7]A point of discontinuity $a \in \mathbb{R}$ of a TTF f is called *positive* if $\lim_{x \to a^-} f(x) < \lim_{x \to a^+} f(x)$ holds ($\lim_{x \to a^-}$ and $\lim_{x \to a^+}$ denote the one-side limit from the left and the right respectively).

authors show that their setup yields an NP-hard problem for all choices of non-negative α and β, as long as $\alpha \neq \beta$ and $\beta > 0$ holds. To do so, they reduce the *number partitioning problem* (see “subset sum problem” in Garey and Johnson [38]). With $\alpha > \beta > 0$, this already implies that the MC queries considered in this thesis are also NP-hard. This thesis reports a translation of their proof into the context of additional time-invariant costs that can be chosen freely from $\mathbb{R}_{\geq 0}$ (see Section 6.1.1).

Time-Dependent Route Planning on Road Networks. We now pass over to the work that has been done on time-dependent road networks. This includes a brief review of the time-dependent versions of some goal-directed speedup techniques for constant travel costs already reviewed in Section 1.3.1. Note that none of these techniques deals with generalized time-dependent travel costs beyond travel times. A comparison with the time-dependent contraction hierarchies (TCHs) described in this thesis can be found in Section 5.6.4.

Relevance of Time-Dependent Route Planning [30]. In a case study Demiryurek et al. investigate whether time-dependent travel times make at all a difference compared to constant travel times: For a road network of California they find that time-dependent routes are 36 % better on average. During rush hours—that is, 7:00 to 9:30 AM and 4:00 to 6:00 PM—this rises to 68 % and 43 % respectively. The similarity (i.e., the percentage of shared edges) of time-dependent routes and constant cost routes is 28 % on average, never exceeding 87 %. A very interesting result is the number of distinct optimal routes over the day: 7 on average, and at most 12. To obtain the TTFs, the authors use a traffic model based on traffic sensor data from highways and arterial streets all over Los Angeles County (together 6 300 traffic sensors). The data has been collected over a period of two years.

The 68 % improvement of time-dependent routes during morning rush hours suggests that the resulting TTFs are relatively strongly varied; at least, in comparison with the TTFs of the German road network data that we use for the experiments in this thesis. There, the TTFs vary up to 36 % on average (see Section 5.6.1). This raises the question, whether the TTFs of Demiryurek et al. model the travel time changes of a single day or, for example, the average travel time changes of midweek traffic over a whole year. In the latter case, TTFs tend to be less varied. Of course, the travel time changes of single day are still relevant for time-dependent route planning in practice; for example, if traffic simulation is constantly performed based on the current traffic situation. Then, it is possible to provide TTFs that predict the travel times of the next few hours. We describe such a scenario at the end of this thesis (see Section 7.3).

Time-Dependent ALT (TD-ALT) [65]. An early attempt to provide fast and exact EA queries is the time-dependent version of the goal-directed technique ALT (see page 34). The lower bound π_t is computed as in case of ALT with constant travel costs, where $\min f_e$ is used as constant travel cost for every edge e. This works, however, not so effective: The difference between values $\min f_e$ used to compute π_t and values $f_e(\tau)$ used during edge relaxation seems to be too large to provide sufficiently tight lower bounds. The resulting speedup compared to time-dependent Dijkstra is only 3.6. Using bidirectional search to some extend makes everything even slower. Allowing inexact results yields better running times with small average error but relatively large maximum error.

Time-Dependent CALT (TD-CALT) [24]. The time-dependent version of CALT (see page 35) works better than of ALT. The query algorithm is a bidirectional search that goes upward, similar as in case of constant travel costs. The backward search works in terms of minima $\min f_e$ of TTFs f_e. The forward search is also allowed to hop downward, but only on the nodes where the backward search reached the core (i.e., the upper level). Within the core TD-ALT is used, again with π_t in terms of minima $\min f_e$. TD-CALT shows reasonable EA query times as well as low preprocessing times and moderate memory consumption. Allowing inexact results decreases the EA query time significantly. As in case of TD-ALT the average error is low, but the maximum error is relatively large.

Time-Dependent SHARC (TD-SHARC) [21]. The most successful goal-directed technique for time-dependent route planning is the time-dependent version of SHARC (see page 35). TD-SHARC shows moderate memory consumption and very good EA query times if time-dependency is weak. If time-dependency is strong, however, EA query times are worse (but still fair). The preprocessing of TD-SHARC takes rather long, and if time-dependency is strong this gets even worse. TD-L-SHARC, a combination with TD-ALT, improves this significantly and yields reasonable running times at the cost of increased (but still moderate) memory consumption. If inexact queries are allowed (*heuristic* variant), TD-SHARC performs much better: Though preprocessing times get even larger, the answering of EA queries gets really fast. Thereby, the maximum error is rather small. Note that TD-SHARC is the only speedup technique besides TCHs that can answer TTP queries. Compared to TCHs, however, this runs pretty slow.

Time-dependency affects the performance of SHARC a lot: EA queries are considerably slower than queries with constant travel costs, and preprocessing takes much longer. This is because arc-flags are very difficult to compute in the time-dependent case. To compute high-quality arc-flags, one would have to perform a great amount of one-to-all backward travel time profile (TTP) searches (see Section 4.3.1), preferably on all levels of the underlying multi-level partition (remember that SHARC recursively partitions the road network into regions making it a multi-level approach). These one-to-all backward searches start at the border nodes of the regions. Of course, this is not feasible. Instead, TD-SHARC performs backward one-to-all Dijkstra-searches in terms of $\min f_e$ and $\max f_e$ for $e \in E$ (*economical* variant), or conservatively approximated backward one-to-all profile searches where the number of interpolation points is fixed to 24 (*generous* variant), or additional exact backward one-to-all profile searches on the highest level (*aggressive* variant). The heuristic variant performs 12 time-dependent backward Dijkstra searches, each with a fixed arrival time, as well as two Dijkstra searches in terms of $\min f_e$ and $\max f_e$ for $e \in E$.

Space Efficient TD-SHARC [14]. Space efficient TD-SHARC reduces the memory consumption of heuristic (i.e., inexact) TD-SHARC significantly at the cost of slower (but still fast) EA querying. To achieve this, three different techniques are used: arc-flags are represented in a compressed way, some shortcuts are omitted, and some shortcuts stored without TTFs. The compressed representation of arc-flags simply maps all occurring flag patterns to a small number of conservative patterns which do not alter the result of the computation but prune the search less effectively (i.e., a lossy but conservative compression). Whenever a shortcut without TTF is relaxed during querying, it is unpacked on-the-fly to enable the evaluation of the omitted TTF. This is quite similar to *unpacking-on-demand* as used by ATCHs (see Section 5.4.2 on page 233). The difference is that ATCHs store approximate versions of the shortcuts' TTFs. Space efficient TD-SHARC omits them completely. Note that exact TD-SHARC already compresses arc-flags (but in a lossless manner). Also, it already omits some shortcuts.

Distributed TCHs [54]. TCHs not only parallelize well on shared memory architectures, but also on distributed memory. To achieve this, Kieritz et al. divide the road network into as many partitions as available processing units (PUs). During preprocessing each PU is responsible for contracting the nodes in its particular partition. To do so, each PU not only maintains its partition, but also a $(\lfloor k/2 \rfloor + 1)$-halo[8] around it to enable witness searches with a hop limit of k (see Section 5.2.2 on page 196). The preprocessing works similar as described in Section 5.2, but before the nodes in the next independent node set can be contracted (also see Section 5.2.2), the newly created shortcut edges must be send to the affected PUs.

During querying, each PU also maintains more than its own partition; namely, the part of the CH that can be reached from the nodes in the partition by searching upward. To answer an EA query with start node s and destination node t, the

[8]The *ℓ-halo* of a subgraph $G' \subseteq G$ is the part of G that is reachable from G' via paths of ℓ or less hops.

two PUs that are responsible for s and t, respectively, each perform a complete upward search starting from s and t. The PU responsible for s then sends the complete search space to the PU responsible for t, which performs the downward search afterwards (for details how TCHs can be used to answer EA queries see Section 5.3.1).

The main benefit of distributed TCHs is that the memory intensive preprocessing of TCHs can be performed on a cluster of cheap PUs with rather little main memory. Though the overall data redundancy is relatively large for distributed TCHs, the data volume arising on a single PU is nevertheless significantly reduced compared to a complete TCH. For querying when main memory is restricted, one could also use ATCHs (see Section 5.4). ATCHs show reduced memory consumption without utilizing distributed memory.

Time-Dependent Many-to-Many [42]. TCHs are, like CHs, not only useful to answer one-to-one queries, but also to perform many-to-many EA and TTP computations. The authors describe five different algorithms that *precompute* different sets of information. One of these algorithms precomputes a table; namely, a full $|S| \times |T|$-table of TTPs. The other four algorithms only compute *some* information that is used to make EA and TTP “lookups” faster. More precisely, given a start node $s \in S$ and a destination node $t \in T$ there is still some work left that has to be done before the resulting EA time or TTP can be returned. This is different from the original many-to-many approach that has been developed for constant travel costs (see the short summary in Section 1.3.1 on page 31).

For exact TTP lookups, only the full table brings a speedup of more than 26 compared to the fastest TCH-based algorithm (namely, exact TTP queries using ATCHs as described in Section 5.4.3). Looking up a TTP in the full table requires no computation at all. However, the full table needs lots of space. For $|S| = |T| = 500$ it needs 27 GiB for example. For exact EA lookups, the full table is also fastest, but the other algorithms, which are more space efficient, show good speedups too; that is, nearly three orders of magnitude further speedup compared to exact TCH-based EA queries.

Note that the results of exact time-dependent many-to-many precomputations are quite space consuming, even if the full table is avoided. That is one of the reasons why the authors consider *inexact* querying, which reduces memory consumption a lot. The running times needed by precomputation and TTP lookups are also reduced. EA lookups, in contrast, do not get much faster. An interesting result is an upper bound of the relative error of inexact travel time computations with piecewise linear functions. It not only depends on the maximum relative error of the involved TTFs but also on the maximum occurring slope of any such TTF. It turns out that the observed error is usually much smaller than this theoretical error bound.

Time-Dependent One-to-All [39]. The algorithmic ingredients used to speed up time-dependent many-to-many computations can also be used to speed up one-to-all TTP queries. In fact, Geisberger combines them with ideas taken from PHAST (see Section 1.3.1 on page 32): He exploits that the downward graph of a TCH is a DAG and utilizes the pruning techniques from time-dependent many-to-many. This ideas, however, are not applied to a complete TCH, but only to the most important 10 000 nodes of a TCH (i.e., a *core* of a TCH). The resulting one-to-all TTP queries on the core take about half a minute of time on average. Compared to 10 000 random one-to-one TTP queries that are performed not only on the core but on the complete hierarchy using ATCHs (see Section 5.4.3), the speedup is one order of magnitude. It also turns out that the one-to-all TTP queries on the core parallelize well in shared memory: The speedup achieved with 8 threads is 5.7. Geisberger suggests that his method could be used to generate high-quality time-dependent arc-flags on the core of a TCH enabling a time-dependent version of CHASE (see Section 1.3.1 on page 36). Although this sounds promising, nobody has tried that so far.

1.3.3 Route Planning with Other Generalized Travel Costs

Besides time-dependent travel costs, there are also other kinds of generalized travel costs that have been considered so far. We only summarize three examples here that all deal with multi-dimensional costs in some way.

Flexible CHs [40]. Flexible CHs answer one-to-one queries if the constant travel costs of edges e are linear combinations of two different kinds of cost; that is

$$c_e = a_e + p \cdot b_e$$

where a_e could be the travel time and b_e the fuel consumption for example. The discrete parameter $p \in \{0, 1, 2, \ldots, N\}$ needs not to be fixed before preprocessing. Instead, users can choose it dynamically just when submitting a query. On the first glance this looks somewhat similar to time-dependent travel times with additional time-invariant costs, but it is different. There, the optimal path depends on the departure time. Here, the optimal path depends on the parameter p.

Adapting CHs to this setup, the main challenge is that more shortcuts must be added during preprocessing to preserve the shortest paths for all possible values of p. The resulting hierarchy would be less sparse which means that memory consumption and query time would increase. To overcome this, the hierarchy is split into multiple hierarchies for different values of p. However, the authors do it in a way that the lower part is equal for all hierarchies. It hence suffices to store the lower part only once. Applying this approach recursively, one gets a *flexible*

CH that is split multiple times in a tree-like manner. Flexible CHs bring good speedups, especially in combination with CALT (see Section 1.3.1 on page 35).

Flexible CHs can also be used to answer *parameter profile queries*: Given a start node s and destination node t one computes a function

$$c_{st} : \{0, 1, 2, \ldots, N\} \to \mathbb{R}_{\geq 0}$$

that maps the parameter p to the corresponding optimal cost for traveling from s to t. If there are k different optimal paths from s to t, then $\mathrm{O}(k)$ one-to-one queries with appropriate values of p in the flexible CH are performed. This works because c_{st} is concave. Note that the witness search works similar, but with multiple plain Dijkstra searches for different values of the parameter p on the respective *remaining graph* (for details of CH construction see Section 2.3.1).

CHs for Multi-Dimensional Costs. Storandt [81] adapts CHs to the two-dimensional *constraint shortest path (CSP)* problem to do route planning for bicycles. The cost of an edge e is a pair $c_e|d_e$, where $c_e \geq 0$ is the distance and $d_e \geq 0$ the difference in elevation. A user query asks for a minimum distance path whose total difference in elevation does not exceed a given bound. To make CHs work in this setup, the witness search does not utilize multi-label search (see Section 2.2.6 for an explanation). This would be too expensive. Instead, Storandt performs multiple conventional Dijkstra searches where edges e have cost

$$L_e := x \cdot c_e + (1 - x) \cdot d_e$$

for different values of a parameter $x \in [0, 1]$, which is similar to witness and profile search as performed by flexible CHs [40]. Again, this works because the parameter cost profiles are concave for every path. Note that some witness paths may be missed. The query algorithm is a bidirectional multi-label search that only goes upward, a preceding conventional CH query suffices if the resulting up-down-path already fulfills the elevation bound. The query times are an improvement compared to multi label search without hierarchy but are still very large. Note that the most important 0.5 % of nodes are not contracted to save preprocessing and query time (there would be too much shortcuts). A combination with arc-flags may bring a further improvement, but Storandt only tries this for small graphs.

Funke and Storandt [37] further generalize this kind of CHs to costs with more than two dimensions. In this setup, however, they only answer *conic combination shortest path* queries exactly. These are the continuous, multi-dimensional version of the queries answered by flexible CHs. CSP queries are only answered approximately. The reason is that the query algorithm uses no longer multi-label search but multiple conic combination shortest path queries for different parameter configurations. These configurations are determined by some kind of multi-dimensional binary search in the parameter space. For more than two dimensions the problem structure gets less convenient. As a consequence, it can happen that no path fulfilling the constraints is found, even if there is one.

For conic combination shortest paths with three dimensional costs, the query times are quite reasonable. With two dimensional costs, they are quite good. For an instance of the Californian road network with about 11 million nodes, three-dimensional approximate CSP queries are answered within less than 1.5 sec, for two dimensions within about 20 msec. The rate of missed paths for three dimensions is 0.05 % and the found paths are not more than 1.04 times worse than the optimum for the Californian instance.

Pareto SHARC [27]. Pareto SHARC generalizes the original SHARC (for a summary see Section 1.3.1 on page 35) to compute a set of Pareto optimal paths (see Section 2.1.3). An arc-flag a_i^e must be *true* if the edge e is part of a Pareto optimal path that leads into region R_i. The authors find that the number of Pareto optimal labels per node can increase dramatically with the size of the road network. Preprocessing and query time can increase even more. To overcome this problem, the authors relax the notion of dominance, both during preprocessing and querying. More precisely, a label $\ell = (C, c_1, \dots, c_k)$ of a node u is also considered to dominate another label $\ell' = (C', c'_1, \dots, c'_k)$ of u if $(1+\varepsilon) \cdot C < C'$ or $\gamma \cdot C/C' < \sum_i c'_i / \sum_i c_i$ holds for two fixed values $\varepsilon \geq 0$, $\gamma > 0$.

This means that of the $k+1$ different travel costs is considered as the leading cost (namely, the cost C). The first condition says that paths are ignored if their leading cost is worse than the leading costs of another path by a factor above $1+\varepsilon$. The second condition (also known as *pricing*) says that paths are ignored if a loss in the leading cost is not justified by a gain in the other costs compared to other paths. With this relaxed notion of dominance and conveniently chosen parameters ε and γ, the authors achieve to compute about 5 different Pareto optimal paths within less than 40 msec for an instance of the Western European road network. With about 5 hours, the preprocessing takes rather long, but this is still tolerable.

1.4 References

Many results presented in this thesis have already been published on conferences [5, 6, 9], in a journal [8], and in an early technical report [7]. All these articles have been published by the author of this thesis together with Daniel Delling, Robert Geisberger, Sabine Neubauer, Peter Sanders, and Christian Vetter in different combinations. Many wordings of these articles have been used or rephrased in the thesis. This is also the case for this first chapter.

Christian Vetter made very important contributions to the TCH preprocessing as described Section 5.2 during his student research project [82] and as a student assistant. Altogether, Vetter is responsible for a substantial part of the preprocessing, both regarding concepts and implementation (for details see Section 5.7).

Sabine Neubauer provided an implementation of the Imai-Iri Algorithm [52] that she prepared during her student research project [66]. It is heavily used by our ATCHs and inexact TCHs, two variants of TCHs discussed in Chapter 5.

1.5 Outline

Chapter 2 introduces basic concepts like real functions, basic graph theory, Pareto optimal paths, intersection of line segments. Also, it summarizes basic data structures and algorithms, including adjacency arrays, priority queues, Dijkstra's algorithm, bidirectional search, A* search, and multi-label search. It also contains an introduction to original CHs.

Chapter 3 considers the basic properties of time-dependent edge weights and the corresponding time-dependent shortest paths. The first part considers the special case that time-dependent travel costs are only travel times. This includes a discussion of travel time functions (TTFs), earliest arrival (EA) paths, basic operations on TTFs including their efficient computation, and convenient rules for calculating with TTFs. The second part essentially provides the analogous discussion for time-dependent travel times with additional travel costs.

Chapter 4 discusses all the basic Dijkstra-like algorithms that are used as ingredients of the route planning techniques described in this thesis. This includes single-label searches in forward and backward direction as well as time-dependent multi-label searches. Chapter 4 also provides proofs of correctness for all these algorithms and discusses other aspects like running time or how to stop them early if they are used to answer one-to-one queries.

Chapter 5 in depth discusses all variants of time-dependent contraction hierarchies (TCHs) that deal with time-dependent travel times as travel costs. This includes the preprocessing, basic earliest arrival (EA) and travel time profile (TTP) querying with TCHs, the pruning technique stall-on-demand, space efficient querying with approximate TCHs (ATCHs), and also inexact TCHs, whose main purpose is to provide even faster TTP queries. All described techniques are evaluated experimentally.

Chapter 6 discusses how TCHs can be adapted to work in the presence of additional time-invariant costs. This includes a discussion of complexity, a discussion why exact preprocessing is problematic with additional costs, and also some preliminary ideas how exact preprocessing may be achievable. Chapter 6 also describes a heuristic preprocessing procedure and a query procedure that would be exact with exact TCH structures. The chapter ends with an experimental evaluation.

Chapter 7 provides a conclusion, a discussion of possible future work, and an outlook how time-dependent route planning techniques could be deployed to become beneficial for a larger number of users.

2

Fundamentals

This chapter briefly summarizes the fundamentals that are needed to understand the contributions made in this thesis. Nothing in this chapter is new. Some of the material is assumed to be known to many readers. This includes real functions, directed graphs, shortest paths, some basic computational geometry, adjacency arrays, priority queues (PQs), Dijkstra's algorithm [33], and A* search [49] (see Sections 2.1.1, 2.1.2, 2.1.4, 2.2.1, 2.2.2, 2.2.3, and 2.2.5). Other things—like Pareto optimal paths, bidirectional search, and multi-label search—are well established but less widely known (see Sections 2.1.3, 2.2.4, and 2.2.6). Pareto optimal paths and multi-label search are described by Hansen [48] and Martins [60]. The other things are either folklore or explained much more detailed in the textbook by Mehlhorn and Sanders [61]. The basic geometry is taken from Cormen et al. [16], Hill [50], and Sedgewick [79].

There is also some material in this chapter (see Section 2.3) which is relatively advanced; namely, the original CHs that provide fast and exact route planning for constant non-negative travel costs. This hierarchical route planning technique is one of the results of the rather recent research on route planning (see Geisberger et al. [43, 44]). It forms the basis of the time-*dependent* contraction hierarchies considered in this thesis. So, we provide a relatively detailed explanation. Many important aspects, however, have been omitted. These things are discussed in the original papers.

A reader that is already familiar with the one or another subject treated in this chapter may skip the respective sections of course.

2.1 Preliminaries

This section covers the most basic notation, definitions and facts. This includes real functions, directed graphs, shortest paths, and Pareto optimal paths.

2.1.1 Real Functions

Most functions that occur in this thesis map real numbers to real numbers. In this case the following three operations are well defined.

1. The *composition* $g \circ f$ (we say g "after" f) of two real functions $f, g : \mathbb{R} \to \mathbb{R}$ is defined by $g \circ f : x \mapsto g(f(x))$.
2. The *pointwise sum* $f + g$ of two real functions $f, g : \mathbb{R} \to \mathbb{R}$ is defined by $f + g : x \mapsto f(x) + g(x)$.
3. The *pointwise minimum* $\min(f, g)$ of two real functions $f, g : \mathbb{R} \to \mathbb{R}$ is defined by $\min(f, g) : x \mapsto \min\{f(x), g(x)\}$.

The set of real functions is closed under these three operations.

Composition, pointwise sum, and pointwise minimum are *associative*. That is, for arbitrary real functions $f, g, h : \mathbb{R} \to \mathbb{R}$ the following equations hold:

$$h \circ (g \circ f) = (h \circ g) \circ f \tag{2.1}$$
$$f + (g + h) = (f + g) + h \tag{2.2}$$
$$\min(f, \min(g, h)) = \min(\min(f, g), h) \tag{2.3}$$

Pointwise sum and pointwise minimum are also commutative:

$$f + g = g + f \tag{2.4}$$
$$\min(f, g) = \min(g, f) \tag{2.5}$$

In general, composition is not commutative. It is also not fully distributive over pointwise sum and minimum, but at least *right-distributive*:

$$(f + g) \circ h = f \circ h + g \circ h \tag{2.6}$$
$$\min(f, g) \circ h = \min(f \circ h,\, g \circ h) \tag{2.7}$$

By $\mathrm{id} : \mathbb{R} \to \mathbb{R}$ we denote the *identity (function)*. It is defined by $\mathrm{id} : x \mapsto x$, which means that every real number is mapped to itself. Composed with other functions, the identity leaves them completely unchanged:

$$f = f \circ \mathrm{id} = \mathrm{id} \circ f \tag{2.8}$$

Note that the identity is the linear function with slope 1 that goes right through the origin of the Cartesian coordinate system (see Figure 2.1). If a function f is a *one-to-one mapping* (i.e., $f(x) = f(y)$ implies $x = y$), then f can be *inverted*.[1] More precisely, there exists a function $f^{-1} : \mathbb{R} \to \mathbb{R}$ with

$$f \circ f^{-1} = f^{-1} \circ f = \mathrm{id}\ . \tag{2.9}$$

[1]One-to-one mappings are often called *injective* mappings.

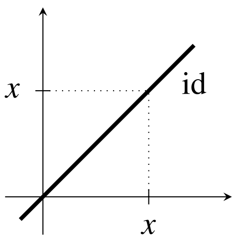


Figure 2.1. The graph of the identity function.

The *inverse function* f^{-1} of a one-to-one mapping f is unique. Geometrically, the graph of the inverse function f^{-1} is just the reflection of the graph of f with the graph of the identity function as axis. If f is no one-to-one mapping, then f^{-1} is defined by

$$f^{-1}(x) := \{y \in \mathbb{R} \mid y = f(x)\} \tag{2.10}$$

as a set-valued function. Note that

$$(g \circ f)^{-1} = f^{-1} \circ g^{-1} \tag{2.11}$$

holds true, regardless whether f and g are one-to-one mappings or not. To understand that consider

$$\begin{aligned} y \in (g \circ f)^{-1}(x) &\Leftrightarrow x = (g \circ f)(y) = g(f(y)) \Leftrightarrow f(y) \in g^{-1}(x) \\ &\Leftrightarrow y \in f^{-1}(g^{-1}(x)) = f^{-1} \circ g^{-1}(x) \ . \end{aligned}$$

A real function $f : D \to \mathbb{R}$ that is only defined on a strict subset of $\mathbb{R}$ (i.e., $D \subseteq \mathbb{R}$ but $D \neq \mathbb{R}$) is called a *partial* real function. We sometimes write $\operatorname{dom} f := D$ to denote the *domain* of f. In case $\operatorname{dom} f = \mathbb{R}$, we call f a *total* real function. In this thesis, we always have $\operatorname{dom} f = \mathbb{R}$ or $\operatorname{dom} f$ is an interval. A real function $f : D \to \mathbb{R}$ is called *continuous* if $f(x_0) = \lim_{x \to x_0} f(x)$ holds for all $x_0 \in \mathbb{R}$. If $\lim_{x \to x_0} f(x)$ does not exist for some $x_0 \in \mathbb{R}$ or if $f(x_0) \neq \lim_{x \to x_0} f(x)$ holds, then x_0 is called a *point of discontinuity* (or *discontinuity* for short) of f. If f only fulfills $f(x_0) = \lim_{x \to x_0^-} f(x)$ for all $x_0 \in \mathbb{R}$, then f is called *left-continuous*. *Right-continuous* functions fulfill the analogous condition. A (point of) discontinuity x_0 is called *positive* if $\lim_{x \to x_0^-} f(x) < \lim_{x \to x_0^+} f(x)$ holds, and *negative* if $\lim_{x \to x_0^-} f(x) > \lim_{x \to x_0^+} f(x)$ holds.

Note that the *left-sided limit* $\lim_{x \to x_0^-} f(x)$ of $f : D \to \mathbb{R}$ is the unique real number $a \in D$ that fulfills the following condition: For all $\varepsilon > 0$ there is $\delta > 0$ such that $|f(x) - a| < \varepsilon$ holds for all $x \in (x_0 - \delta, x_0)$. The *right-sided limit* $\lim_{x \to x_0^+} f(x)$ is defined analogously.

Lemma 2.1. *The composition of left-continuous functions is again left-continuous.*

Proof. A possible proof for continuous functions utilizes convergent sequences (e.g., see Sections 6.4 and 6.7 in Walter's textbook [84]). It can easily be adapted to work with left-continuous functions. □

2.1.2 Directed Graphs and Shortest Paths

Given a finite set V and a subset $E \subseteq V \times V$ we call $G = (V, E)$ a *directed graph*. The elements of V are the *nodes* and the elements of E the *edges* of G. We usually write $u \to v$ to denote an edge $(u, v) \in E$. Edges $u \to v$ of G often have a non-negative real value $c \in \mathbb{R}_{\geq 0}$ assigned, which we call *edge cost* or *travel cost*. We often write $u \to_c v$. If we reverse all edges of G we obtain the *transpose* graph $G^\top$ of G defined by

$$G^\top := \left(V, E^\top\right) \quad \text{with} \quad E^\top := \{v \to u \,|\, u \to v \in E\} \;, \tag{2.12}$$

which fulfills $(G^\top)^\top = G$ of course.

A sequence of $k-1$ edges $\langle u_1 \to u_2, u_2 \to u_3, \ldots, u_{k-1} \to u_k\rangle$ of G, such that the *target* node u_i of an edge $u_{i-1} \to u_i$ is the *source* node of the succeeding edge $u_i \to u_{i+1}$, is called a *path* in G. We write $P = \langle u_1 \to u_2 \to \cdots \to u_k\rangle$ for short. The number of edges of P is denoted by $|P|$. It is also called the number of *hops* of P. A subpath $\langle u_1 \to \cdots \to u_i\rangle$ of P is called a *prefix (path)* of P, a subpath $\langle u_i \to \cdots \to u_k\rangle$ of P is called a *suffix (path)* of P. If there is another path $P' = \langle v_1 \to \cdots \to v_\ell\rangle$ such that $u_k = v_1$, then P' can be *appended* to P. This results in the *concatenated* path

$$PP' := \langle u_1 \to \cdots \to u_k = v_1 \to \cdots \to v_\ell\rangle \;. \tag{2.13}$$

We also need the *k-hop neighborhood* of a node u in a graph G defined by

$$\begin{aligned} N_G^k(u) := \big\{v \in V \setminus \{u\} \;\big|\; & G \text{ contains an undirected path} \\ & \text{from } u \text{ to } v \text{ or } v \text{ to } u \text{ of } k \text{ or less hops}\big\} \;. \end{aligned} \tag{2.14}$$

A path $\langle u \to \cdots \to u\rangle$, where start and destination node are the same, is called a *cycle*.

The graph formed by all paths from a node s to a node t in a subgraph $U \subseteq G$ is denoted by $\mathscr{P}_U(s,t)$. For sets of nodes $S, T \subseteq V$, we generalize this notation to

$$\mathscr{P}_U(S,T) := \bigcup_{s \in S, t \in T} \mathscr{P}_U(s,t) \;. \tag{2.15}$$

Note that we use the symbol "$\cup$" not only to denote the union of sets but also of graphs in this thesis. For $S = \{s\}$ or $T = \{t\}$, respectively, we simply write

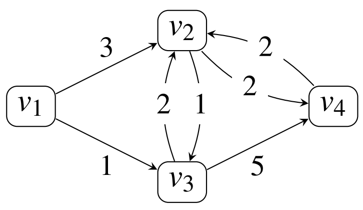


Figure 2.2. A simple directed graph with constant edge costs that contains two shortest paths from v_1 to v_4; namely, $\langle v_1 \rightarrow v_2 \rightarrow v_4 \rangle$ and $\langle v_1 \rightarrow v_3 \rightarrow v_2 \rightarrow v_4 \rangle$. They both have total cost 5.

$\mathscr{P}_U(s,T)$ or $\mathscr{P}_U(S,t)$. We mainly use this to denote *corridors* and *cones*. Given a subgraph $U \subseteq G$, an *opening cone* is a subgraph of U that contains all paths from a node s to the nodes $T \subseteq V$; that is, $\mathscr{P}_U(s,T)$. Analogously, a *closing cone* contains all paths from the nodes $S \subseteq V$ to a node t in U; that is, $\mathscr{P}_U(T,t)$. Note that $\mathscr{P}_U(S,t) \subseteq U \subseteq G$ is a closing cone if, and only if, $\mathscr{P}_{U^\top}(t,S) \subseteq U^\top \subseteq G^\top$ is an opening cone. Moreover, we have

$$\mathscr{P}_{U^\top}(S,T) = \left(\mathscr{P}_U(T,S)\right)^\top . \tag{2.16}$$

A *corridor* is the special case that both S and T are singleton sets; that is, a subgraph $\mathscr{P}_U(s,t)$.

If G has edge costs, then a path $P = \langle u_1 \rightarrow_{c_1} u_2 \rightarrow_{c_2} \cdots \rightarrow_{c_{k-1}} u_k \rangle$ in G has the *(total) cost* $c_P := c_1 + c_2 + \cdots + c_{k-1}$. Since all edge costs are non-negative, there is a minimum possible total cost that a path from a node s to a node t in G can have; namely, the value

$$\mu_G(s,t) := \min\left(\left\{C_{P'} \,\middle|\, P' \text{ is a path from } s \text{ to } t \text{ in } G\right\} \cup \{\infty\}\right) . \tag{2.17}$$

We call $\mu_G(s,t)$ the *minimum (constant) travel cost* from s to t in G. A path P from s to t with $c_P = \mu_G(s,t)$ is called a *shortest path* from s to t. In general there can be multiple shortest paths from s to t. Figure 2.2 shows an example. All subpaths of shortest paths are shortest paths themselves. If all edge costs in G are positive, then a shortest path can not contain any cycles. Note that $\mu_G(s,t) = \infty$ means that G does not contain any path from s to t. We say t is not *reachable* from s in G.

A subgraph $T = (V_T, E_T) \subseteq G$ with a distinguished node $r \in V_T$ is called a *directed tree*[2] with *root r* if T has no cycles and all nodes $u \in V_T \setminus \{r\}$ have exactly one incoming edge in E_T (see Section 4.9 in [55]). As it is well-known, directed trees can also be characterized by paths.

Lemma 2.2 ([55]). *The following two statements are equivalent: (i) The subgraph $T \subseteq G$ is a directed tree with root node r. (ii) There is exactly one path in T from r to all nodes $u \in V_T \setminus \{r\}$.*

[2]also known as *arborescence*

If all the unique paths from the root r to the other nodes $u \in V_T \setminus \{r\}$ are shortest paths in G, then T is called a *shortest path tree* in G.

2.1.3 Pareto Optimal Paths

Pareto optimality[3] is a generalized notion of optimality that can be used in the context of multidimensional costs. Consider the set of real ℓ-tuples $\mathbb{R}^\ell$. We say that $(x_1,\ldots,x_\ell) \in \mathbb{R}^\ell$ *(weakly) dominates* $(y_1,\ldots,y_\ell) \in \mathbb{R}^\ell$ if $x_1 \le y_1 \wedge \cdots \wedge x_\ell \le y_\ell$ holds. If there is also at least one $i \in \{1,\ldots,\ell\}$ with $x_i < y_i$, then we speak of *strict dominance*. A set $M \subseteq \mathbb{R}^\ell$ is called a *Pareto set* (or sometimes a *Pareto optimum*) if no ℓ-tuple in M is dominated by another ℓ-tuple in M. These concepts can be applied to directed graphs which results in a generalized notion of a shortest path [48, 60]. Note that we restrict ourselves to $\ell = 2$ in this thesis.

Consider a graph G where each edge $u \rightarrow_{c|d} v$ has *two* non-negative values $c, d \in \mathbb{R}_{\ge 0}$ assigned. Note that we usually write $c|d$ for two-dimensional costs (c,d). Then, the total cost of a path $P := \langle u_1 \rightarrow_{c_1|d_1} \cdots \rightarrow_{c_{k-1}|d_{k-1}} u_k \rangle$ is two-dimensional and amounts to

$$c_P \,|\, d_P \;:=\; c_1 + \cdots + c_{k-1} \,|\, d_1 + \cdots + d_{k-1} \,. \tag{2.18}$$

The notion of dominating paths only makes sense for paths with the same start and destination node. We say a path P from s to t *(weakly) dominates* a path P' from s to t if $c_P|d_P$ weakly dominates $c_{P'}|d_{P'}$. *Strict dominance* of paths is defined analogously. A path P from s to t in G is called *Pareto optimal* if it is not strictly dominated by any other path from s to t—or, more precisely, if the following condition holds:

$$\forall \text{ paths } P' \text{ from } s \text{ to } t : \big(c_{P'} \,|\, c_{P'} \ne c_P \,|\, d_P \implies c_{P'} > c_P \text{ or } d_{P'} > d_P\big) \tag{2.19}$$

In case of two-dimensional costs one often speaks of *bicriteria shortest paths*.

Hansen [48] shows that exponentially many Pareto optimal paths can occur in bicriteria settings. To do so he uses the construction in Figure 2.3. We recapitulate his argument in the following. The graph in Figure 2.3 contains $2k+3$ nodes and 2^{k+1} different paths from v_0 to v_{k+1}. All these 2^{k+1} paths are Pareto optimal. To understand that consider two different paths P and P' from v_0 to v_{k+1} with total costs $c_P|d_P$ and $c_{P'}|d_{P'}$ respectively. We see that $c_P + d_P = 2^{k+1} - 1 = c_{P'} + d_{P'}$ holds. But then $c_P|d_P$ can not strictly dominate $c_{P'}|d_{P'}$. Assume otherwise that w.l.o.g. $c_P < c_{P'}$ and $d_P \le d_{P'}$ holds. But then we have $c_P + d_P < c_{P'} + d_{P'}$, which is a contradiction. The case $c_P = c_{P'}$ and $d_P = d_{P'}$ is also not possible for $P \ne P'$ as no two paths from v_0 to v_{k+1} have the same total two-dimensional cost. In other

[3]Named after the Italian sociologist and economist Vilfredo Federico Pareto (1848–1923).

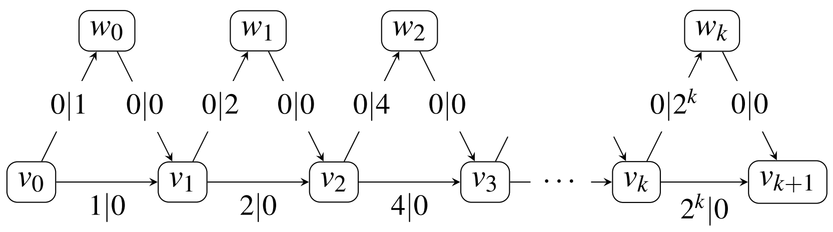


Figure 2.3. The graph used by Hansen [48] to show that exponentially many Pareto optimal paths can occur in bicriteria settings. It has $2k+3$ nodes and contains 2^{k+1} paths from v_0 to v_{k+1} that all have different costs not dominating each other.

words, a graph with $n := 2k+3$ nodes can have $2^{k+1} = 2^{1+(n-3)/2} = \Omega(2^{n/2}) = \Omega(1.41^n)$ Pareto optimal paths between two nodes.

Subpaths of Pareto optimal paths are Pareto optimal paths themselves [60]. Otherwise, a Pareto optimal path $P = \langle s \to \cdots \to t\rangle$ could be decomposed into subpaths $P = RS$ with $R = \langle s \to \cdots \to u\rangle$ such that R is strictly dominated by another path R' from s to u. So, we had $c_{R'} \le c_R$, $d_{R'} \le d_R$, and $c_{R'} + d_{R'} < c_R + d_R$ implying $c_{R'S} = c_{R'} + c_S \le c_R + c_S = c_P$, $d_{R'S} = d_{R'} + d_S \le d_R + d_S = d_P$, and

$$c_{R'S} + d_{R'S} = c_{R'} + c_S + d_{R'} + d_S < c_R + c_S + d_R + d_S = c_P + d_P\ .$$

This means the concatenation $R'S$ would strictly dominate P, a contradiction.

2.1.4 Points and Line Segments

In Section 1.1.2 we say that time-dependent costs are modeled as functions; namely, as TTFs and TCFs. In this thesis, such functions are modeled using piecewise linear functions (see Section 3.1.1 and 3.2.2). The algorithms that deal with piecewise linear functions (see Section 3.1.4) require basic computational geometry in two dimensions, which we explain in this section.

Two basic concepts of geometry are *points* and *line segments*. A points is written as a pair $(x,y) \in \mathbb{R}^2$, a line segment between the points $p,q \in \mathbb{R}^2$ is written as $(pq) \subseteq \mathbb{R}^2$. Note that we consider the end points p and q as excluded from the line segment; that is, $p,q \notin (pq)$. The set of common points of two line segments $(pq),(p'q')$ is denoted by $(pq) \cap (p'q')$. So, $|(pq) \cap (p'q')| = 1$ means that the two segment have exactly one point in common. We call this point (which has to be different from p,q,p',q') the *intersection point*.

To determine whether two line segments intersect, we use the *perp dot product* (see Hill [50]) of two points (x,y) and (x',y') defined as

$$(x,y) \bullet (x',y') := xy' - x'y\ . \tag{2.20}$$

With this product we can determine the orientation of three points in the plane. More precisely, to find out whether three points $p,q,r \in \mathbb{R}^2$ of an ordered sequence $\langle p,q,r \rangle$ are located clockwise, collinear, or counterclockwise relative to each other, we use the predicate *ccw* (see Chapter 24 in Sedgewicks' textbook [79]) defined as

$$ccw(p,q,r) := \begin{cases} 1 & \text{if } \langle p,q,r \rangle \text{ is oriented clockwise} \\ 0 & \text{if } p,q,r \text{ are collinear} \\ -1 & \text{if } \langle p,q,r \rangle \text{ is oriented counterclockwise} \end{cases} . \tag{2.21}$$

Note that Sedgewick uses a slightly different definition of *ccw* that only returns zero if r lies on the line segment (pq). Cormen et al. (see Section 33.1 in their textbook [16]) explain how the perp dot product can be used to obtain the orientation of three points; that is, to calculate *ccw*.

Lemma 2.3 ([16]). *Given $p,q,r \in \mathbb{R}^2$, we have*

$$ccw(p,q,r) = \operatorname{sgn}\big((q-p) \bullet (r-p)\big) , \tag{2.22}$$

where sgn *is the sign function.*[4]

Note that the '$-$' in Equation (2.22) is a component-by-component operation on points. Sedgewick explains, how *ccw* can be used to determine whether two line segments intersect; that is, to check the condition $|(pq) \cap (p'q')| = 1$.

Lemma 2.4 ([79]). *Two line segments $(pq),(p'q')$ without end points intersect in exactly one point if, and only if, $ccw(p,q,p') \cdot ccw(p,q,q') < 0$ and $ccw(p',q',p) \cdot ccw(p',q',q) < 0$ holds.*

Note that Sedgewick discusses a slightly different version of this statement, because of his different definition of *ccw* and because he considers intersection of line segments including end points.

To actually obtain the single intersection point r of (pq) and $(p'q')$ (provided that there is one), we make again use of the perp dot product, as Hill [50] suggests.

Lemma 2.5 ([50]). *The intersection point of two line segments $(pq),(p'q')$ with $|(pq) \cap (p'q')| = 1$ is*

$$r = p + \frac{(p'-q') \bullet (p'-p)}{(p'-q') \bullet (q-p)} \cdot (q-p) . \tag{2.23}$$

In the above equation, '$+$' and '$-$' are component-by-component operations on points.

[4] $\operatorname{sgn}(x)$ yields 1 or -1 for positive or negative x respectively. For $x = 0$ it yields 0.

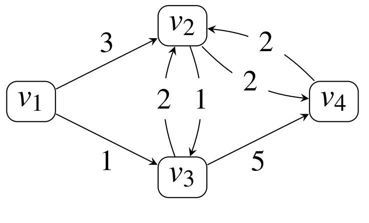
2.2 Basic Data Structures and Algorithms

In this section we recapitulate the most basic algorithmic ingredients that are fundamental for this thesis. This includes Dijkstra's well known algorithm and some of its variants—bidirectional search, A* search, and multi-label search—as well as adjacency arrays and priority queues (PQs), two very basic data structures.

2.2.1 Adjacency Arrays

An *adjacency array* is a simple data structure that represents a directed graph $G = (V,E)$ (e.g., see Section 8.2 in the textbook by Mehlhorn and Sanders [61]). It needs $\Theta(|V| + |E|)$ space and iterating over all outgoing edges $u \to v \in E$ of a node $u \in V$ can be done in $O(1)$ time per edge.

Every node $u \in V$ and every edge $u \to v \in E$ is identified by a unique *node id* from $\{0, \dots, |V| - 1\}$ and a unique *edge id* from $\{0, \dots, |E| - 1\}$ respectively. So, every node $u \in V$ and every edge $u \to v \in E$ can be identified with its id, which is as good as to say $u \in \{0, 1, \dots, |V| - 1\}$ and $u \to v \in \{0, 1, \dots, |E| - 1\}$ respectively. An array *targetNodeId* of size $|E|$ contains exactly one entry for every edge $u \to v \in E$; namely, the entry *targetNodeId*$[u \to v]$. It stores the node id i_v of the target node v. All edges $u \to v_1, \dots, u \to v_k$ with common source node u have *consecutive* edge ids. So, the array *targetNodeId* is partitioned into $|V|$ subarrays, each containing the target node ids of the outgoing edges for each node in V. For every node u we store the smallest edge id of its outgoing edges in an array *firstEdgeId* of size $|V| + 1$; namely, at the entry *firstEdgeId*$[u]$. Hence, the subarray of *targetNodeId* that corresponds to the node u reaches from the index *firstEdgeId*$[u]$ to the index *firstEdgeId*$[u+1] - 1$. The last entry *firstEdgeId*$[|V|]$ does not correspond to any existing node of G but is necessary to determine the



	0	1	2	3	4
firstEdgeId	0	2	4	6	7

	0	1	2	3	4	5	6
targetNodeId	1	2	3	2	1	3	1

	0	1	2	3	4	5	6
cost	3	1	2	1	2	5	2

Figure 2.4. The simple directed graph from Figure 2.2 (left) represented by an adjacency array (right). The nodes v_1, v_2, v_3, and v_4 have the node ids 0, 1, 2, and 3 respectively. The small numbers denote the indices of the array entries and correspond to node and edge ids respectively. With *firstEdgeId*$[1] = 2$ and *firstEdgeId*$[2] = 4$, for example, we know that the edge ids of the outgoing edges of v_2 are 2 and 3 and that the subarray *targetNodeId*$[2..3]$ contains the target nodes of the respective edges. They entry *cost*$[3] = 1$ says that edge $v_2 \to v_3$ has cost 1.

end of the last subarray of *targetNodeId*.

Additional information about nodes and edges can be stored in further arrays of size $|V|$ and $|E|$ respectively. If G has edge costs, for example, the costs can be stored in an array *cost* of size $|E|$. Figure 2.4 shows a simple example of an adjacency array representation of a directed graph with constant edge costs.

2.2.2 Priority Queues

A *priority queue (PQ)* Q is a fundamental data structure to store elements that are prioritized with *keys* (e.g., see Chapter 6 in the textbook by Mehlhorn and Sanders [61]). More precisely, Q stores a subset of $\mathit{Elements} \times \mathit{Keys}$ where the set *Elements* defines the possible data elements and *Keys* is a *totally ordered set*. The set *Keys* is called totally ordered if there is a relation "$\preceq$" that is

- *antisymmetric* (i.e., $x \preceq y$ and $y \preceq x$ implies $x = y$ for all $x, y \in \mathit{Keys}$),
- *transitive* (i.e., $x \preceq y$ and $y \preceq z$ implies $x \preceq z$ for all $x, y, z \in \mathit{Keys}$), and
- *total* (i.e., for all $x, y \in \mathit{Keys}$ we have $x \preceq y$ or $y \preceq x$).

Totally ordered sets are also *reflexive*; that is, $x \preceq x$ holds for all $x \in \mathit{Keys}$. Reflexivity follows directly from totality. Note that PQs also work if "$\preceq$" is not antisymmetric. But this is never the case in this thesis. A PQ Q must support the four following operations:

1. $Q.\mathit{insert}(e : \mathit{Elements}, x : \mathit{Keys})$. Adds the pair $(e, x) \in \mathit{Elements} \times \mathit{Keys}$ to the set stored by Q. Usually we require that no pair (e, x') is already present in Q.
2. $Q.\mathit{deleteMin}() : \mathit{Elements}$. Removes a pair (e, x) from Q such that x is *minimal* amongst all keys in Q. More precisely, for all (e', x') in Q we have $x \preceq x'$. If (e_0, x_0) is the removed pair, then e_0 is given back as return value.
3. $Q.\mathit{updateKey}(e : \mathit{Elements}, x : \mathit{Keys})$. Replaces the pair (e, x') in Q by the pair (e, x). If no pair (e, x') is present in Q, then this operation is illegal.
4. $Q.\mathit{min}()$. Returns the currently minimal key in Q and ∞ if Q is empty.

Note that *updateKey* is often referred to as *decreaseKey*. In this case, the new key x must be "smaller" than the old key x'; that is, $x \preceq x'$ must hold. In route planning keys are often non-negative numbers $q, r \in \mathbb{R}_{\geq 0}$ with $q \preceq r$ if, and only if, $q \leq r$. Sometimes, however, $q \preceq r$ is equivalent to $q \geq r$, as in case of latest departure interval search (see Section 4.3.4, Algorithm 4.6). To express that, we use the operations *deleteMax* instead of *deleteMin* and call Q a *maximum PQ*.

There are many different realizations of the abstract data type PQ. These are, for example, *binary heaps* and *Fibonacci heaps*. Binary heaps are relatively simple and provide logarithmic asymptotic running time for all the operations except for $\mathit{min}()$, which takes constant time. Fibonacci heaps have better asymptotic running time, but they are more complicated and usually not so fast in practice.

2.2.3 Dijkstra's Algorithm

Dijkstra's algorithm[5] [33] is a well-known method to compute shortest paths in a directed graph $G = (V, E)$ with non-negative constant edge costs (e.g., see Section 10.3 and 10.4 in the textbook by Mehlhorn and Sanders [61]). Given a start node $s \in V$, we want to compute the minimum cost $\mu_G(s,u)$ for traveling from s to u for every $u \in V$ as well as a corresponding shortest path P_u. To do so, we maintain a tentative cost $d[u] \in \mathbb{R}_{\geq 0}$, which we call a *node label*, and a tentative predecessor information $p[u] \in V$ for all $u \in V$. Initially, we set $d[u] := \infty$ and $p[u] := \perp$ for all nodes $u \in V$; only for the start node s, we set $d[s] := 0$. The symbol $\perp$ means that a node has currently no predecessor. Every node has one of two possible states, either *settled* or *unsettled*. Initially, all nodes are unsettled.

During execution we successively *settle* one node after another. More precisely, we always choose an unsettled node u with minimal tentative cost; that is, a node $u \in \arg\min\{d[v] \mid v \in V \text{ is unsettled}\}$. Having chosen such a node u we *relax* all its outgoing edges $u \to_c v \in E$. This means that for each such edge we check whether $d[u] + c$ is smaller than $d[v]$ and if so, we set $d[v] := d[u] + c$ and $p[v] := u$. So, we update the tentative cost and the tentative predecessor information of the target node v whenever we find a better path to v. When all outgoing edges of u are relaxed, we consider u as settled. The execution of Dijkstra's algorithm stops when all nodes being reachable from s are settled. This is the case when all unsettled nodes have tentative cost ∞.

Dijkstra's algorithm obeys some monotonicity property: If a node u is settled before a node v, we always have $d[u] \leq d[v]$ after u is settled. Also, as soon as u settled, we have $d[u] = \mu_G(s,u)$ (see Section 10.3 in [61]). So, as Dijkstra's algorithm never increases $d[u]$, no node is ever settled twice. This is actually where the term "settling" comes from: As soon as the state of a node u changes from unsettled to settled, we know that $d[u]$ has reached $\mu_G(s,u)$ and cannot be decreased anymore.

A shortest path from s to u is given implicitly by the predecessor information as soon as u is settled. More precisely, a shortest path from s to a settled node u can be constructed starting from u by successively looking up the predecessor of each node until we arrive at s. The resulting shortest path is of the form

$$P_u = \langle s = d[\ldots d[d[u]] \ldots] \to \cdots \to d[d[u]] \to d[u] \to u \rangle \ . \tag{2.24}$$

This only works if u is reachable from s of course. Today, Dijkstra's algorithm is usually formulated using a PQ (see Section 2.2.2). Algorithm 2.1 shows the

[5]Named after the Dutch computer scientist Edsger W. Dijkstra (1930–2002).

Algorithm 2.1. The one-to-one version of Dijkstra's algorithm. For a given start node s and a destination node t that is reachable from s, the algorithm returns a shortest path from s to t. If t is not reachable from s, it returns the empty path $\langle\rangle$. The pseudocode is mainly adopted from Mehlhorn and Sanders [61].

```
1  function dijkstra(s,t : V) : Path
2      d[u] := ∞, p[u] := ⊥ for all u ∈ V, d[s] := 0
3      Q : PriorityQueue
4      Q.insert(s, 0)
5      while Q ≠ ∅ do
6          u := Q.deleteMin()                              // settle node u
7          if u = t then return ⟨s → ··· → d[d[t]] → d[t] → t⟩
8          foreach u →_c v ∈ E do                  // relax outgoing edges of u
9              if d[u] + c < d[v]  then
10                 if d[v] = ∞ then Q.insert(v, d[u] + c)
11                 else Q.updateKey(v, d[u] + c)
12                 d[v] := d[u] + c
13                 p[v] := u
14     return ⟨⟩                                    // t not reachable from s
```

respective pseudo code. There, the settled nodes are exactly the nodes u fulfilling

$$d[u] < \infty \quad \text{and} \quad u \notin Q\ .$$

Thus, a node is settled by removing it from the PQ (see Line 6). As nodes are never settled twice, no node is ever reinserted into the PQ by Dijkstra's algorithm.

In route planning we are usually *not* interested in shortest paths from s to *all* other nodes $u \in V$. Instead, we often look for a shortest path from s to one given node t. For such one-to-one queries we do not have to execute Dijkstra's algorithm completely, but we can stop it earlier. In fact we can stop as soon as the node t is settled (see Line 7). If Line 7 is omitted, the algorithm computes a shortest path from s to all nodes that are reachable from s.

It is important to note that the predecessor information p, which is computed by Dijkstra's algorithm, implicitly represents a subgraph of G; namely, the graph $G(p) := (V(p), E(p))$ with

$$V(p) := \{u \in V \mid p[u] \neq \emptyset \text{ or } u \in p[v] \text{ with } v \in V\}\ , \tag{2.25}$$

$$E(p) := \{u \to v \in E \mid u = p[v]\}\ . \tag{2.26}$$

Note that $G(p)$ as defined by Equation (2.25) and (2.26) is a tree because of the condition in Line 9. We call $G(p)$ the *predecessor tree* represented by p. After termination of Dijkstra's algorithm, $G(p)$ is even a shortest path tree in G. Note that we generalize the notion of a predecessor tree to the notion of a predecessor graph in Section 4.1.2.

2.2.4 Bidirectional Search

The fact that we are mainly interested in one-to-one queries from a start node s to a destination node t enables us to apply Dijkstra's algorithm in a bidirectional manner (e.g., see Section 10.8 in the textbook by Mehlhorn and Sanders [61]). The idea behind this *bidirectional search* is to run two instances of Dijkstra's algorithm "at the same time": a *forward search* starting from s and a *backward search* starting from t. Both searches can be stopped when they meet somewhere in the middle. More precisely, we stop them as soon as a node has been settled by both searches.

Algorithm 2.2 shows a possible pseudocode for bidirectional search. We can see that forward and backward search are not really performed at the same time but in an alternating manner. The variable Δ stores the current search direction: $\Delta = s$ for forward and $\Delta = t$ for backward search. The search direction is flipped in every step of the while loop (see Line 8). Both searches maintain their own tentative costs and predecessor information as well as their own PQ. In case of the forward search these are d_s, p_s, and Q_s, in case of the backward search d_t, p_t, and Q_t. The backward search relaxes all edges in reverse direction. Actually, the backward search is an instance of Dijkstra's algorithm running on the transpose graph $G^\top$. This is reflected by the fact that E_t is identified with $E^\top$ (see Line 16).

Note that the first node settled by both searches is not necessarily part of a shortest path. To obtain $\mu_G(s,t)$ and a corresponding shortest path from s to t, we have to inspect some more nodes w that have been reached by both searches. While doing so, we choose a node that minimizes $d_s[w] + d_t[w]$ (see Line 12). It is not necessary to inspect all nodes that have been reached by both searches. According to Mehlhorn and Sanders it suffices to inspect the reached but unsettled nodes with respect to the forward search as well as the first node settled by both searches (see Section 10.8 in [61]).

On road networks, bidirectional search runs roughly two times faster than the unidirectional one-to-one version of Dijkstra's algorithm (according to experiments by Bauer et al. [11]). This is because the Dijkstra search spreads out in a more or less circular manner and the number of settled nodes is somehow proportional to $\mu_G(s,t)^2$ hence. But forward and backward search of the bidirectional search meet somewhere in the middle. As a consequence the radius of the circular area is about half as big each. But then, they each settle a number of nodes that is roughly proportional to $\mu_G(s,t)^2/4$. So, together they settle a number of nodes roughly proportional to $\mu_G(s,t)^2/2$. This is illustrated in Figure 2.5.

Algorithm 2.2. Bidirectional Dijkstra search. Both searches have their own PQ and maintain their own tentative cost and tentative predecessor information. In case of the forward search, there are Q_s, d_s, and p_s respectively. In case of the backward search, there are Q_t, d_t, and p_t respectively. The variable $\Delta \in \{s,t\}$ stores the current search direction.

```
function bidirectionalDijkstra(s,t : V) : Path
    $d_s[u] := d_t[u] := \infty$, $p_s[u] := p_t[u] := \perp$ for all $u \in V$
    $d_s[s] := d_t[t] := 0$
    $Q_s, Q_t$ : PriorityQueue
    $Q_s.insert(s,0)$, $Q_t.insert(t,0)$
    $\Delta := t$                                          // current search direction
    while $Q_s \neq \emptyset$ or $Q_t \neq \emptyset$ do
        $\Delta := \neg\Delta$                                   // with $t = \neg s$ and $s = \neg t$
        if $Q_\Delta = \emptyset$ then $\Delta := \neg\Delta$
        $u := Q_\Delta.deleteMin()$
        if $d_{\neg\Delta}[u] < \infty$ and $u \notin Q_{\neg\Delta}$ then       // u also settled by other search?
            Choose $v \in \arg\min \{ d_s[w] + d_t[w] \mid w \in Q_s \cup \{u\} \}$
            $P_s := \langle s \to \cdots \to d_s[d_s[v]] \to d_s[v] \to v \rangle$
            $P_t := \langle v \to d_t[v] \to d_t[d_t[v]] \to \cdots \to t \rangle$
            return $P_s P_t$
        foreach $u \to_c v \in E_\Delta$ do                   // where $E_s = E$ and $E_t = E^\top$
            if $d_\Delta[u] + c < d_\Delta[v]$ then
                if $d_\Delta[v] = \infty$ then $Q_\Delta.insert(v, d_\Delta[u] + c)$
                else $Q_\Delta.updateKey(v, d_\Delta[u] + c)$
                $d_\Delta[v] := d_\Delta[u] + c$
                $p_\Delta[v] := u$
    return $\langle\rangle$
```

2.2.5 A* Search

The purpose of the bidirectional search presented in the section before is to provide an algorithm for one-to-one queries that is faster than the slow one-to-one version of Dijkstra's algorithm. However, the achieved speedup of roughly two is by far not sufficient. Another way to provide faster one-to-one queries is to apply *goal-direction*. A well-known example of this approach is *A* search* [49] (e.g., see Section 10.8.1 in the textbook by Mehlhorn and Sanders [61]).

The idea of goal-direction is to bias the search in a way that it spreads out rather towards the destination node t than in a circular manner (see Figure 2.5 for an illustration). In case of A* search we achieve this bias by using a *potential*

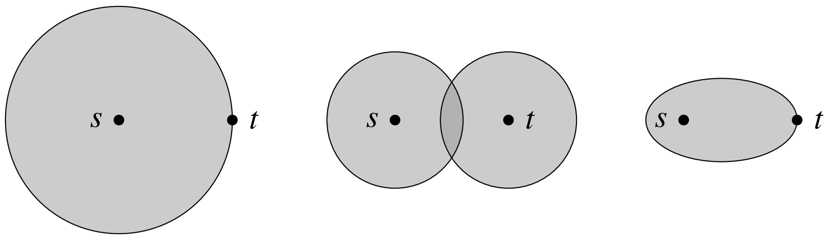


Figure 2.5. Mehlhorn and Sanders [61] as well as Schultes [78] illustrate how the search on a graph spreads out for different algorithms: Dijkstra's algorithm (left), bidirectional search (middle), and A* search (right).

function $\pi_t(u)$ that conforms to the following two conditions:

$$\pi_t(u) \leq c + \pi_t(v) \quad \text{for all} \quad u \rightarrow_c v \in E \tag{2.27}$$

$$0 \leq \pi_t(u) \leq \mu_G(u,t) \quad \text{for all} \quad u \in V \tag{2.28}$$

Having such a potential function at hand, A* search is just a very slight modification of Dijkstra's algorithm: We simply use $d[u] + \pi_t(u)$ instead of $d[u]$ as key when inserting a node u into the PQ. This is actually the same as running Dijkstra's algorithm on a graph with transformed edge costs. The transformation can be done by replacing the cost c of an edge $u \rightarrow_c v$ with

$$c' := c + \pi_t(v) - \pi_t(u)\,. \tag{2.29}$$

It is important to note that the shortest paths in the graph are not changed by this transformation (e.g., see Lemma 10.9 in the textbook by Mehlhorn and Sanders [61]). This is because of the well-known fact that the sum that makes up the transformed travel cost c'_P of a path $P = \langle u_1 \rightarrow_{c_1} \cdots \rightarrow_{c_{k-1}} u_k \rangle \subseteq G$ telescopes; that is,

$$c'_P = \sum_{i=1}^{k-1} \big(c_i + \pi_t(u_{i+1}) - \pi_t(u_i)\big) = \pi_t(u_k) - \pi_t(u_1) + \sum_{i=1}^{k-1} c_i\,, \tag{2.30}$$

which means the transformation of a path's cost solely depends on its start and destination node. Also, because of Condition (2.27) the transformed edge costs from Equation (2.29) are never negative. As a consequence, Dijkstra's algorithm can really be applied after the transformation. Moreover, from Condition (2.28) we know that $\pi_t(t) = 0$ holds, which implies $d[t] = \mu(s,t)$ is fulfilled at the moment when t is removed from the PQ and the algorithm is stopped (see Line 7 of Algorithm 2.1). Altogether, we see that A* search really returns a shortest path for a given start node s and destination node t.

The purpose of $\pi_t(u)$ is to estimate the minimum travel cost $\mu_G(u,t)$ to t from all $u \in V$. The smaller the difference $\mu_G(u,t) - \pi_t(u)$ is for all $u \in V$, the less nodes should be settled by the A* search before it stops. To understand that, compare the PQ key of a node u that lies on a shortest path from s to t and of a node v that lies apart such a shortest path: If $\mu_G(w,t) - \pi_t(w)$ is small for most nodes $w \in V$, then it is likely that

$$d[u] + \pi_t(u) \approx \mu_G(s,u) + \mu_G(u,t) \leq \mu_G(s,v) + \mu_G(v,t) \approx d[v] + \pi_t(v)$$

holds for the final values of $d[u]$ and $d[v]$, which means that u is very probably settled before v. So, the nodes on shortest paths from s to t are more likely to be settled than the other nodes. But as soon as t is settled we are finished.

To understand why π_t is called a potential function, we again assume that $\mu_G(w,t) - \pi_t(w)$ is small for most nodes $w \in V$. Then, it is likely that $\mu_G(u,t) > \mu_G(v,t)$ implies $\pi_t(u) > \pi_t(v)$, which means that the transformed cost $c' = c + \pi_t(v) - \pi_t(u)$ of an edge $u \rightarrow_c v$ fulfills $c' < c$. Analogously, $\mu_G(u,t) > \mu_G(v,t)$ suggests $c' > c$. This means that the transformed costs of edges that "point towards" t tend to be smaller and of the other edges tend to be larger. So, π_t actually characterizes a warp of the plain[6] such that t lies at the bottom of a valley. An A* search is a search on this warped plain which is biased towards the valley and thus towards t.

A good potential function must fulfill two requirements: On the one hand, it should reduce the number of settled nodes a lot. On the other hand, it should be evaluable fast. If both is fulfilled, one-to-one queries can be answered much faster than with Dijkstra's algorithm.

2.2.6 Multi-Label Search

Multi-label search [48, 60] is a generalization of Dijkstra's algorithm (see Section 2.2.3) to compute the set of all Pareto optimal paths from a start node s to all reachable nodes (see Section 2.1.3). Remember that we only consider bicriteria shortest paths in this thesis. The structure of multi-label search is similar to Dijkstra's algorithm, but there are some remarkable differences.

Instead of a single tentative travel cost and a single tentative predecessor information we maintain a *label set* $L[u]$ for each node $u \in V$. Every *label* $\ell \in L[u]$ of a node u corresponds to a path P_ℓ from the start node s to the node u and is of the form $\ell = (i, u, \gamma|\delta, i')$ where

- $i \in \mathbb{N}$ is a unique id of the label ℓ,
- $\gamma|\delta = c_{P_\ell}|d_{P_\ell}$ is the two-dimensional total cost of P_ℓ, and
- i' is the unique id of the *preceding label* $\ell' = (i', u', \gamma'|\delta', i'') \in L[u']$.

[6]Road networks can be viewed as nearly planar graphs that are "embedded" into the plain.

Algorithm 2.3. Extracts the path that corresponds to the label with the given id i.

```
function $extractPathFromLabelId(i : LabelId) : Path$
    $P := \langle\rangle$
    while $true$ do
        let $(i, u, \gamma|\delta, i')$ be the label with id $i$
        if $i' = \perp$ then break
        let $(i', u', \gamma'|\delta', i'')$ be the label with id $i'$
        append $u' \to u$ to the front of $P$
        set $i := i'$
    return $P$
```

Algorithm 2.4. Computes the set of all Pareto optimal paths from s to t. To extract the Pareto optimal paths that correspond to the labels of t, Algorithm 2.3 is invoked.

```
function $multiLabelSearch(s, t : V) : Set$
    $L[u] := \emptyset$ for all $u \in V$
    $Q := \emptyset : PriorityQueue$
    $Q.insert\big((0, s, 0|0, \perp), 0\big)$
    $L[s] := \{(0, s, 0|0, \perp)\}$
    $i_{\text{next}} := 1$                                    // the next unused label id
    while $Q \neq \emptyset$ do
        $(i, u, \gamma|\delta, i') := Q.deleteMin()$
        if $\gamma + \delta > \max\big\{\gamma^* + \delta^* \,\big|\, (\cdot, t, \gamma^*|\delta^*, \cdot) \in L[t]\big\}$ then break
        foreach $u \to_{c|d} v \in E$ do                      // relax outgoing edges of u
            $\ell_{\text{new}} := (i_{\text{next}}, v, \gamma + c|\delta + d, i)$
            if $\ell$ is not dominated by any label in $L[v]$ then
                $Q.insert(\ell_{\text{new}}, \gamma + c + \delta + d)$
                remove all labels dominated by $\ell_{\text{new}}$ from $L[v]$ and $Q$
                add $\ell_{\text{new}}$ to $L[u]$
                $i_{\text{next}} := i_{\text{next}} + 1$
    $M := \emptyset$                                         // build up set of Pareto optimal paths
    foreach $(i, u, \gamma|\delta, i') \in L[t]$ do
        add the path $extractPathFromLabelId(i)$ to $M$
    return $M$
```

For $u = s$ there is no preceding label and we have $i' = \perp$. For $u \neq s$, in contrast, the preceding label $\ell' = (i', u', \gamma'|\delta', i'')$ has to exist and the condition

$$\gamma|\delta \;=\; \gamma' + c \,\big|\, \delta' + d \tag{2.31}$$

must be fulfilled with $P_\ell = \langle s \to \cdots \to u' \to_{c|d} u \rangle$. All this implies that P_ℓ can be extracted from the labels by successively looking up the preceding label starting from u until we encounter the special symbol $\perp$. Algorithm 2.3 shows pseudocode for this extraction process. The condition in Line 5 holds true if, and only if, $u = s$.

Multi-label search, in contrast to Dijkstra's algorithm, does not settle nodes but labels. Settling a label $(i, u, \gamma|\delta, i')$ means that all outgoing edges of u are relaxed. This is similar to settling a node u as Dijkstra's algorithm does it. Note that the elements that we put into the PQ are also labels instead of nodes. As key of a label $\ell = (i, u, \gamma|\delta, i')$ we could use the two-dimensional cost $\gamma|\delta$ together with the lexicographic order (Martins [60] does this for example). But in our experience, using the sum $\gamma + \delta$ as key runs faster on road networks.

Algorithm 2.4 shows pseudocode for the one-to-one version of multi-label search that computes the set of all Pareto optimal paths from a start node s to a destination node t, but ruling out paths with the same two-dimensional costs. The notion of dominance is extended from the context of two-dimensional costs to the context of labels (e.g., see Line 12). We say, that a label $(i, u, \gamma|\delta, j)$ of a node u is (weakly) dominated by a another label $(i', u, \gamma'|\delta', j')$ of u, if $\gamma' \leq \gamma$ and $\delta' \leq \delta$ hold. If additionally $\gamma' < \gamma$ or $\delta' < \delta$ holds, we speak of strict dominance. Note that *updateKey* is never invoked. Instead, the PQ must support the removal of arbitrary elements (Line 14). Similar to Dijkstra's algorithm, multi-label search shows monotonous behavior: If label $(i_u, u, \gamma_u|\delta_u, j_u)$ is settled before a label $(i_v, v, \gamma_v|\delta_v, j_v)$, then we have $\gamma_u + \delta_u \leq \gamma_v + \delta_v$. As a consequence, the algorithm can be stopped as soon as the current minimum key of the PQ exceeds the maximum key amongst the labels of the target node t (Line 9).

Note that we also could formulate multi-label search in a way that we put nodes into the PQ instead of labels. In this case we use the minimum key amongst all unsettled labels in $L[u]$ as key of a node u. Whenever a label of a node u is settled, we invoke *updateKey* for this node. As soon as the last node in $L[u]$ is settled, u is removed from the PQ by invoking *deleteMin*. However, this formulation of multi-label search is more complicated, so we opted to describe the simpler one. But it must be noted that the implementation of time-dependent multi-label search (see Section 4.4.1) that we use in our experiments (see Section 6.4) puts nodes into the PQ instead of labels.

2.3 CHs with Constant Travel Costs

In this section we give an introduction to the aforementioned original CHs [44] (also see Section 1.1.4 as well as Section 1.3.1 on page 30), an algorithmic technique for fast and exact route planning with constant travel costs. As already said, they form the basis of the *time-dependent* CHs (TCHs) described in this thesis, which generalize CHs to deal with time-dependent travel costs. In Section 2.3.1 we explain the expensive *preprocessing* stage, where we construct the CH structure representing the road network. In Section 2.3.2 we describe the *querying* stage, where we use the precomputed CH structure for the fast and exact answering of one-to-one queries.

2.3.1 Preprocessing

CHs exploit the inherent hierarchical structure of road networks. Some parts of a road network G have usually much higher capacity than others, which naturally corresponds to the routes chosen by many drivers. Accordingly, all nodes of G have to be *ordered* by some notion of *importance* with more important nodes higher up in the hierarchy. The idea is roughly that a node is more important the more shortest paths run over it. So, before we are able to construct the hierarchy, we have to derive an appropriate *node order* first. That is why preprocessing consists of two phases: the *node ordering* and *hierarchy construction*.

Node Ordering. The choice of the node order has great influence on the performance of CHs. A good node order results in a CH which is *flat* and *sparse*. Using a flat and sparse CH to answer one-to-one queries is fast because the query algorithm of CHs is a bidirectional Dijkstra search that only goes upward (for details see Section 2.3.2). A bad node ordering, in contrast, results in a CH which is so high or so dense that bidirectional upward searches take too much time. Finding a good node order is not a trivial task.

In this thesis, however, we do not explain the node ordering techniques developed for the original CH with constant travel costs, but only how the hierarchy construction works. This is because of two reasons:

1. The node ordering techniques of original CHs are not necessary to provide a basic understanding how the preprocessing step of CH works.
2. Later in this thesis we describe other node ordering techniques which are more suited for *time-dependent* road networks.

The node ordering techniques of the original CHs are discussed by Geisberger et al. [43, 44].

In the rest of this section we assume that the node ordering has already happened and that the resulting node order is a total order "$\prec$" (see Section 2.2.2) of the set V. There, $u \prec v$ means, that u is less important than v or, in other words, that v is higher up in the hierarchy than u. Note that total orders are also reflexive; that is, $u \prec u$ holds for all $u \in V$. Reflexivity follows directly from totality.

Hierarchy Construction. Conceptually, the hierarchy consists of $|V|$ stacked *overlay graphs*. An overlay graph of a graph $G = (V,E)$ is a graph $G' = (V',E')$ with $V' \subseteq V$ such that $\mu_{G'}(s,t) = \mu_G(s,t)$ for all $s,t \in V'$ [51]. In other words, all the travel costs between the nodes of G' are the same as in G.

We construct the CH bottom up by successively *contracting* one node after another from the least important to the most important node; that is, in the order given by "$\prec$".[7] Contracting a node x of a graph $G = (V,E)$ means, that we remove x and all its incident edges from G to obtain an overlay graph $G' = (V',E')$ with $V' := V \setminus \{x\}$. However, in an overlay graph all minimum travel costs between the remaining nodes must be preserved. So, if the removal of a path $\langle u \rightarrow_c x \rightarrow_d v\rangle$ increases the minimum travel cost from u to v, then we must add an artificial *shortcut* (edge) $u \rightarrow_{c+d} v$ to E'. Otherwise, some minimum travel costs in G' would be greater than in G. When adding the shortcut $u \rightarrow_{c+d} v$ to E', it can happen that an edge $u \rightarrow_{c'} v$ is already present. We replace the old edge $u \rightarrow_{c'} v$ by the new shortcut edge in this case.

Let $\overline{G} := (\overline{V},\overline{E}) := (V \setminus \{x\}, \{u \rightarrow v \in E \mid u,v \neq x\})$ be the graph we get when we remove x and its incident edges from G. Then,

$$E_x \;:=\; \left\{u \rightarrow_{c+d} v \;\middle|\; \langle u \rightarrow_c x \rightarrow_d v\rangle \text{ is path in } G \text{ and } c+d < \mu_{\overline{G}}(u,v)\right\} \quad (2.32)$$

is the set of shortcuts that we add when x is contracted (we say the shortcuts in E_x are *necessary*). This means that the edge set E' of the overlay graph has the following form:

$$E' \;=\; E_x \cup \left\{u \rightarrow_c v \in \overline{E} \;\middle|\; u \rightarrow_{c'} v \notin E_x\right\} \quad (2.33)$$

Note that a shortcut $u \rightarrow_{c+d} v$ is added to E_x if, and only if,

- the corresponding path $\langle u \rightarrow_c x \rightarrow_d v\rangle$ is a shortest path in G and
- there is no shortest path from u to v in G that lies completely in $\overline{G}$.

We say that the shortcut $u \rightarrow_{c+d} v$ *represents* the path $\langle u \rightarrow_c x \rightarrow_d v\rangle$. Figure 2.6 shows an example how an overlay graph G' is constructed from a very simple graph G. The following lemma shows that the construction of G' as given by the Equations (2.32) and (2.33) really yields an overlay graph.

[7]As the node order "$\prec$" is a total order, we can write down the set V in the form $x_1 \prec x_2 \prec \cdots \prec x_{|V|}$ with $\{x_1,\ldots,x_{|V|}\} = V$. We contract the least important node x_1 first. Then we continue with x_2, x_3, and so on until we finish $x_{|V|}$.

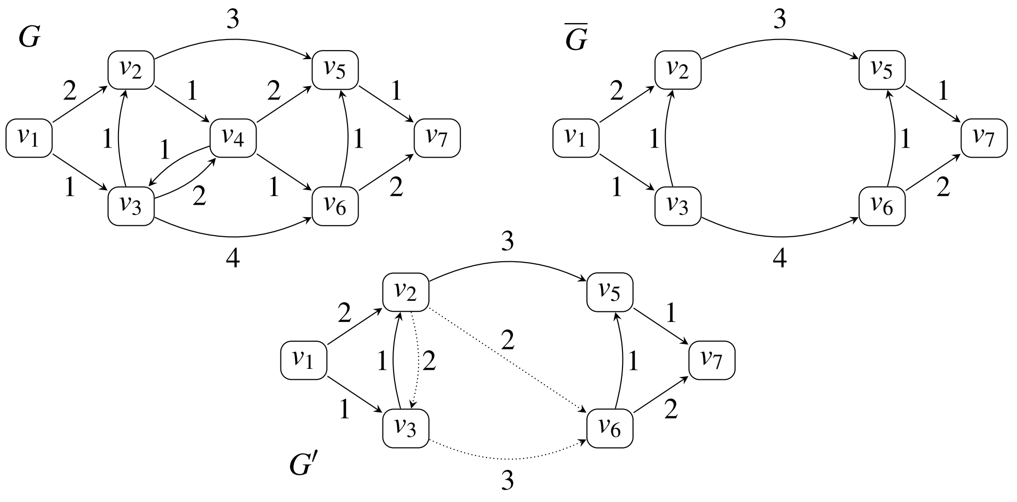


Figure 2.6. An example graph G where we want to contract the node v_4 (top left). Removing v_4 and all its incident edges yields the graph $\overline{G}$ (top right). The obtain an overlay graph G' (bottom), where the travel costs between all nodes except for v_4 are the same as in G, we construct G' as determined by the Equations (2.32) and (2.33). All shortcuts are displayed dotted. Note that G contains the shortest path $\langle v_3 \rightarrow_2 v_4 \rightarrow_2 v_5 \rangle$, which is removed together with v_4, but there is no shortcut $v_3 \rightarrow_4 v_5$ in G' representing this path. This is because $\overline{G}$ still contains another shortest path from v_3 to v_5. So, the shortcut $v_3 \rightarrow_4 v_5$ is not necessary. Also note that the shortcut $v_2 \rightarrow_3 v_6$ replaces the edge $v_3 \rightarrow_4 v_6$. Otherwise, the travel cost from v_3 to v_6 would not be preserved but increased on the contraction of v_4.

Lemma 2.6. *Contracting a node x in a graph G preserves all travel costs between the remaining nodes. That is, $\mu_{G'}(s,t) = \mu_G(s,t)$ holds for all $s,t \in V' = V \setminus \{x\}$.*

Proof. Consider a path P' from s to t in G'. Every edge of P' is either contained in $\overline{G} \subseteq G$ or a newly inserted shortcut edge from E_x. Every such shortcut edge $u \rightarrow_{c+d} v$ corresponds to some path $\langle u \rightarrow_c x \rightarrow_d v \rangle \subseteq G$. We just replace these shortcuts by the corresponding subpaths and get a path from s to t in G which has the same total cost as P'. So, $\mu_{G'}(s,t) \geq \mu_G(s,t)$ holds for all $s,t \in V$.

Now, consider a path P from s to t in G with $s,t \neq x$. If P contains a subpath $\langle u \rightarrow_c x \rightarrow_d v \rangle$, then we have $c + d \geq \mu_{\overline{G}}(u,v)$ or E_x contains a shortcut edge $u \rightarrow_{c+d} v$. In both cases we can remove the subpath $\langle u \rightarrow_c x \rightarrow_d v \rangle$ from P and replace it by a path in G' which has a total cost less or equal $c+d$. It can also happen that P contains an edge $u \rightarrow_c v \in \overline{E}$, but $u \rightarrow_{c'} v \in E_x$ with $c > c'$ holds. Then, we can reduce the edge cost of $u \rightarrow v$ from c to c'. Altogether, there is a path from s to t in G' which has no greater total cost than $P \subseteq G$. So, $\mu_{G'}(s,t) \leq \mu_G(s,t)$ holds for all $s,t \in V$. □

Contracting all nodes of a road network G from the least important to the most important one, we get a hierarchy of overlay graphs $G_1, G_2, \ldots, G_{|V|}$ with $G_1 = G$. There, $G_{i+1} = (V_{i+1}, E_{i+1})$ is an overlay graph of $G_i = (V_i, E_i)$ for $1 \leq i < |V|$. Remember that we only contract one node to obtain the next overlay graph. So, we have $|V_{i+1}| = |V_i| - 1$, $|V_{|V|}| = 1$, and $E_{|V|} = \emptyset$. Of course, it would need very much memory to store all the graphs $G_1, \ldots, G_{|V|}$ at the same time. However, the CH-based algorithm for one-to-one queries (see Section 2.3.2) suffices with a far more compact representation. In fact, we store the hierarchy of graphs in a very condensed way: Every node is materialized exactly once and the *original edges* of the graph G are put together with the artificial shortcut edges. So, we have the graph $H := (V, E_H)$ with

$$E_H := \Big\{ u \to_c v \;\Big|\; u \to v \in \bigcup\nolimits_{i=1}^{|V|} E_i \text{ and } c = \min\big\{ c' \mid u \to_{c'} v \in E_i \text{ for some } i \big\} \Big\} . \tag{2.34}$$

Note that an edge $u \to v$ can occur in multiple sets E_i with $1 \leq i \leq |V|$, sometimes with different costs. This happens, for example, if $u \to_c v$ is already present in G but with $c > \mu_G(s,t)$. In this case we choose the minimal possible cost.

An additional information that we store together with H is whether an edge leads upward or downward in the hierarchy. With "$\prec$" being the node order, we call $u \to_c v$ an *upward edge* if $u \prec v$ holds, and a *downward edge* if $v \prec u$ holds. We also store information about the paths that are represented by every shortcut. Consider a shortcut $u \to_{c+d} v$ representing the path $\langle u \to_c x \to_d v \rangle$. Then we annotate the shortcut $u \to_{c+d} v$ with the *middle node* x. The middle node is enough information to *expand* a shortcut to the path it represents (see Section 2.3.2). Note that the graph H—together with the annotated middle nodes and the information whether an edge leads upward or downward—is in fact all information that the CH query needs. It is thus the final result of the preprocessing and what we call a CH structure.

Algorithm 2.5 shows pseudocode summarizing the hierarchy construction. The CH $H = (V, E_H)$ is constructed from G by successively adding shortcuts to E_H. To do so, the nodes of the given road network (V, E) are successively contracted in the order $x_1 \prec x_2 \prec \cdots \prec x_{|V|}$. During construction, the algorithm maintains the *remaining graph* $R = (V_R, E_R)$ that consists of all nodes not yet contracted as well as all original edges and shortcuts between these nodes. After removing the current node x_i (see Line 7), the remaining graph R acts as $\overline{G}$ as on page 68. Then, after adding or replacing all necessary shortcut edges respectively (see Line 11 and 12), R acts as overlay graph G' with edge set E' as in Equation (2.33). Line 11 ensures that the travel costs of a shortcut is always minimal, as required by Equation (2.34). The annotated middle node is always updated to-

Algorithm 2.5. Given a graph (V,E) with non-negative constant travel costs as well as a node order "$\prec$" this algorithm constructs a CH.

```
function constructCh((V,E) : Graph, ≺ : NodeOrder) : Graph
    $E_H := E$
    $(V_R,E_R) := (V,E)$                    // the "remaining graph" $R = (V_R,E_R)$
    sort $V$ according to "$\prec$" yielding $x_1 \prec x_2 \prec \cdots \prec x_{|V|}$
    for $i = 1$ to $|V|$ do
        $M :=$ all paths $\langle u \rightarrow_c x_i \rightarrow_d v\rangle$ in $(V_R,E_R)$
        remove $x_i$ and all its incident edges from $(V_R,E_R)$      // at this point $R = \overline{G}$
        foreach path $\langle u \rightarrow_c x_i \rightarrow_d v\rangle$ in $M$ do
            mark $u \rightarrow x_i$ as downward and $x_i \rightarrow v$ as upward edge in $E_H$
            if $c + d < \mu_{(V_R,E_R)}(u,v)$ then               // see Equation (2.32)
                remove $u \rightarrow v$ from $E_H$ and $E_R$ if present
                add $u \rightarrow_{c+d} v$ to $E_H$ and $E_R$
                annotate $u \rightarrow_{c+d} v$ with the middle node $x_i$
        // at this point $R = G'$
    return $(V, E_H)$
```

gether with a shortcuts travel cost and always corresponds to the minimal current travel cost hence (see Line 13).

To check the condition in Line 10 one can perform a one-to-one Dijkstra search from u to v in R. A shortest path different from $\langle u \rightarrow_c x_i \rightarrow_d v\rangle$ found by this Dijkstra search is called a *witness path*, as it proofs that $\langle u \rightarrow_c x_i \rightarrow_d v\rangle$ is not a unique shortest path from u to v and the shortcut can be omitted hence. The one-to-one Dijkstra search is called *witness search* accordingly. Note that, in reality, witness search not only consists of a simple one-to-on Dijkstra search. In fact, several sophisticated optimizations are applied to make the CH construction as fast as possible (see Section 5.2.5 on page 207).

A complete preprocessing would include the node ordering phase of course. But as said before, we do not explain the node ordering for constant travel costs in this thesis. We close this section with a basic property of CHs.

Lemma 2.7. *Let H be a CH constructed from a road network $G = (V,E)$. Then, we have $G \subseteq H$ and $\mu_H(s,t) = \mu_G(s,t)$ for all $s,t \in V$.*

Proof. According to Equation (2.34), H contains every edge that occurs in any of $E_1,\ldots,E_{|V|}$ (though with minimal occurring travel cost each). So, with $E_1 = E$ we have $G \subseteq H$ and this implies $\mu_H(s,t) \leq \mu_G(s,t)$ for all $s,t,\in V$. But $\mu_H(s,t) < \mu_G(s,t)$ can never be true. This is because every edge in H that is not present in G

is taken from an overlay graph of G. But this means that none of these edges can introduce a smaller cost for traveling from one node to another. $\square$

2.3.2 Querying

We already said that a one-to-one query with a CH is a bidirectional Dijkstra search that only goes upward. More precisely, the forward search only relaxes upward edges. The backward search only relaxes downward edges but in reverse direction. We define $H_\uparrow$ and $H_\downarrow$ to be the subgraphs of H that only consist of upward and downward edges respectively:

$$\begin{aligned} H_\uparrow &:= (V, E_\uparrow) := (V, \{u \to_c v \in E_H \mid u \prec v\}) \\ H_\downarrow &:= (V, E_\downarrow) := (V, \{u \to_c v \in E_H \mid v \prec u\}) \end{aligned} \tag{2.35}$$

We call $H_\uparrow$ and $H_\downarrow$ the *upward graph* and the *downward graph* respectively. So, the forward search is a Dijkstra search in $H_\uparrow$ and the backward search in $H_\downarrow^\top$. Because of the fact that $\prec$ is a total order (see page 58), we know that $H_\uparrow$ and $H_\downarrow$ are edge disjoint DAGs. That is, $H_\uparrow$ and $H_\downarrow$ do not contain any cycles and $E_\uparrow \cap E_\downarrow = \emptyset$ holds. A path

$$P := \langle u_1 \to \cdots \to u_k = x = v_1 \to \cdots \to v_\ell \rangle$$

with $u_1 \prec \cdots \prec u_k$ and $v_\ell \prec \cdots \prec v_1$ is called an *up-down-path* with *top node* x. Obviously, we have $\langle u_1 \to \cdots \to u_k \rangle \subseteq H_\uparrow$ and $\langle v_1 \to \cdots \to v_\ell \rangle \subseteq H_\downarrow$; that is, the prefix path $\langle u_1 \to \cdots \to u_k \rangle$ goes only upward and the suffix path $\langle v_1 \to \cdots \to v_\ell \rangle$ goes only downward. If P is also a shortest path, then it is called a *shortest up-down-path*.

It must be noted that a shortest path in $H_\uparrow$ is not necessarily a shortest path in H. The same holds analogously for $H_\downarrow$. This raises the question whether bidirectional upward search really yields a shortest path in H. This, however, follows from the guaranteed existence of shortest up-down-paths, which is ensured by the following lemma.

Lemma 2.8 ([44]). *Let H be a CH constructed from a road network $G = (V, E)$ and $s, t \in V$ such that t is reachable from s in G. Then, there is a shortest up-down-path from s to t in H.*

Proof. We recapitulate the proof by Geisberger et al. [44], because it provides a good understanding of how CHs work.

Consider a shortest path $P := \langle s = u_1 \to_{c_1} u_2 \to_{c_2} \cdots \to_{c_{k-1}} u_k = t \rangle$ in the original road network $G \subseteq H$. Further consider the *inner local minima* of this path and choose the smallest one amongst them. More precisely, choose the *inner*

Algorithm 2.6. Query algorithm of CHs with constant non-negative travel costs. Given a CH structure H as well as a start node s and a destination node t, this algorithm computes a shortest up-down path from s to t in H. The pseudocode is similar to the one given by Geisberger et al. [44].

```
function $chQuery(s,t : V) : Path$
    $d_s[u] := d_t[u] := \infty$, $p_s[u] := p_t[u] := \perp$ for all $u \in V$
    $d_s[s] := d_t[t] := 0$
    $Q_s, Q_t : PriorityQueue$
    $Q_s.insert(s,0)$, $Q_t.insert(t,0)$
    $\Delta := t$                                      // current search direction
    $x := \perp, B := \infty$ // top node and total cost of the best up-down-path found so far
    while $(Q_s \neq \emptyset$ or $Q_t \neq \emptyset)$ and $\min\{Q_s.min(), Q_t.min()\} \leq B$ do
        $\Delta := \neg\Delta$                          // with $t = \neg s$ and $s = \neg t$
        if $Q_\Delta = \emptyset$ then $\Delta := \neg\Delta$
        $u := Q_\Delta.deleteMin()$
        if $d_s[u] + d_t[u] < B$ then $x := u$, $B := d_s[u] + d_t[u]$
        foreach $u \rightarrow_c v \in E_\Delta$ do              // where $E_s = E_\uparrow$ and $E_t = E_\downarrow^\top$
            if $d_\Delta[u] + c < d_\Delta[v]$ then
                if $d_\Delta[v] = \infty$ then $Q_\Delta.insert(v, d_\Delta[u] + c)$
                else $Q_\Delta.updateKey(v, d_\Delta[u] + c)$
                $d_\Delta[v] := d_\Delta[u] + c$
                $p_\Delta[v] := u$
    $P_\uparrow := \langle s = p_s[\dots p_s[x]\dots] \rightarrow \cdots \rightarrow p_s[x] \rightarrow x\rangle$
    $P_\downarrow := \langle x = p_t[\dots p_t[t]\dots] \rightarrow \cdots \rightarrow p_t[t] \rightarrow t\rangle$
    return $P_\uparrow P_\downarrow$
```

node $u_i \neq s,t$ with $u_i \prec u_{i-1}, u_{i+1}$, such that all other inner nodes $u_j \neq s,t$ with $u_j \prec u_{j-1}, u_{j+1}$ fulfill $u_i \prec u_j$. At the time when u_i was contracted during the CH construction (see Algorithm 2.5), there has either been a shortest path P_{uv} from u_{i-1} to u_{i+1} in (V_R, E_R) or a shortcut $u_{i-1} \rightarrow_{c_{i-1}+c_i} u_{i+1}$ (which is also a shortest path) has been added to E_H. In either case we replace the subpath $\langle u_{i-1} \rightarrow_{c_{i-1}} u_i \rightarrow_{c_i} u_{i+1}\rangle$ by P_{uv} or the shortcut respectively. This yields a new shortest path P' from s to t in H where all inner local minima y fulfill $u_i \prec y$. So, applying this construction repeatedly finally terminates because V is a finite set. The resulting path is a shortest path without any inner local minimum. So, the resulting path is a shortest up-down-path from s to t in H. □

Lemma 2.8 tells us that the CH query, which is actually a bidirectional upward search, surely finds a shortest up-down-path in H. Algorithm 2.6 shows the respective pseudocode. In fact, the pseudocode is quite similar to the pseudocode of a bidirectional Dijkstra search without hierarchy (see Algorithm 2.2). The only differences are that the forward and backward search are done in $H_\uparrow$ and $H_\downarrow$ respectively and that the while loop can *not* be stopped as soon as the first node is settled by both searches. Instead, the algorithm maintains the top node x and the total cost B of the best up-down-path found so far. The execution can be stopped as soon as the minimum keys in *both* PQs exceed B (see Line 8).

It is important to note that the shortest up-down-paths computed by Algorithm 2.6 contain shortcuts, which are artificial edges that are not present in the original road network. Hence, they can not be used directly to generate driving directions. In Section 2.3.1, however, we said that every shortcut $u \rightarrow_{c+d} v$ represents a path $\langle u \rightarrow_c x \rightarrow_d v \rangle$ and that $u \rightarrow_{c+d} v$ is annotated with the middle node x. So, $u \rightarrow v$ can be *expanded* to the represented path. If $u \rightarrow x$ or $x \rightarrow v$ is a shortcut itself, it can be expanded recursively. Because of the fact that "$\prec$" is a total order, this process surely terminates and results in a path P from u to v that has total cost $c+d$ and completely lies in G. This idea can be applied to whole up-down-paths of course.

Corollary 2.9 ([44]). *Let H be a CH constructed from a road network G and $P \subseteq H$ an up-down-path from s to t. Expanding all shortcuts in P recursively yields a path $P' \subseteq G$ from s to t with the same total travel cost.*

Especially, expanding a shortest up-down-path yields a shortest path in the original road network. So, one-to-one queries can be answered by using Algorithm 2.6 and expanding the resulting shortest up-down-path. Using the annotated middle nodes to expand up-down-paths recursively, is an important concept in the context of CHs. Given an up-down-path $P = \langle s \rightarrow \cdots \rightarrow t \rangle \subseteq H$, we call the fully expanded version $P' = \langle s \rightarrow \cdots \rightarrow t \rangle \subseteq G$ of P the *original path* in G *represented* by P.

So far, we only considered how the query algorithm works and why it is correct. More precisely, we argued that a bidirectional upward Dijkstra search finds a shortest up-down-path, which is guaranteed to exists by the CH construction, and that expanding a shortest up-down-path yields a corresponding shortest path in G. In other words, we have shown that CHs provide exact answering of one-to-one queries. However, a good route planning technique must also provide *fast* answering. CHs, in fact, provide very fast answering. Originally, this is only shown experimentally, but there are also some theoretical analyses meanwhile [2, 62].

Experiments show that the search spaces of forward and backward search are very small in practice. The reason is that well-constructed CH structures are *flat* and *sparse*. The former means, that all purely upward paths and all purely downward paths in a CH structure have quite a limited number of hops. The latter means, that most nodes in a CH structure have not too many incident upward edges. Both conditions help to keep search spaces of bidirectional Dijkstra small, which results in small running times.

2.4 References

Nothing in this chapter is new. Except for Pareto optimal paths and multi-label search, all these things are either quite common or explained in much more detail in the textbook by Mehlhorn and Sanders [61]. Pareto optimal paths and multi-label search are described by Hansen [48] and Martins [60] for example. The basic geometry is taken from Cormen et al. [16], Hill [50], and Sedgewick [79]. Contraction hierarchies (CHs) [43, 44] are a well established algorithmic framework for route planning with constant travel costs.

3

Time-Dependent Road Networks

This chapter provides a detailed description of time-dependent road networks as they are understood in this thesis. We already explained in the introduction (see Section 1.1.2) that time-dependent road networks are modeled as directed graphs $G = (V, E)$. A route in the road network corresponds to a path in G and vice versa. Every edge $u \to v \in E$ has two functions $f : \mathbb{R} \to \mathbb{R}_{\geq 0}$ and $C : \mathbb{R} \to \mathbb{R}_{\geq 0}$ assigned: the *travel time function (TTF)* f and the *travel cost function (TCF)* C. The TTF f specifies the *travel time* $f(\tau)$ one needs to get from the node u to the node v via the edge $u \to v$ when starting at *departure time* τ. Similarly, the TCF C specifies the *travel cost* $C(\tau)$ one must pay to get from the node u to the node v via the edge $u \to v$ with departure time τ. We usually write $u \to_{f|C} v$.

TTFs are modeled as periodic continuous piecewise linear (p.w.l.) functions. Moreover, all TTFs fulfill the *FIFO property*; that is, $\tau < \tau'$ implies $f(\tau) + \tau \leq f(\tau') + \tau'$. The FIFO property means that a later departure never results in an earlier arrival. TCFs are relatively restricted in this thesis, as G is only allowed to have TCFs of the form $C : \tau \mapsto f(\tau) + c$ where $c \in \mathbb{R}_{\geq 0}$ depends on the edge $u \to v$ but not on the departure time. That is why we speak of time-dependent travel times with *additional time-invariant costs*. Note that we also consider the case that c is time-dependent. This is necessary for one of the route planning techniques discussed in this thesis, because it needs such generalized additional costs to work properly.

In Section 3.1 we take a closer look at TTFs and at paths that are optimal with respect to travel time only. In Section 3.2 we consider additional time-invariant and additional time-dependent costs as well as the respective optimal paths.

3.1 Time-Dependent Travel Times

This subsection considers the properties of TTFs (see Section 3.1.1) and also of three basic operations that we define on TTFs (see Section 3.1.2). These operations enable us to define paths with minimum time-dependent travel times conveniently. Such paths are, in fact, the optimal time-dependent paths if travel time is the only cost that arises during traveling; that is, if $C = f$ holds for all edges $u \rightarrow_{f|C} v \in E$ (see Section 3.1.3). Finally, we explain how the three basic operations on TTFs can be computed efficiently (see Section 3.1.4). This is important, because it affects the performance of the route planning algorithms described in this thesis.

Much of the material discussed in this section is not new or relatively straightforward. Especially the time-dependent minimum travel time paths explained in Section 3.1.1 are already discussed by Dean [17]. We nonetheless consider these things in a detailed way, because they are fundamental to the time-dependent route planning techniques described in this thesis (for details see Section 3.3).

3.1.1 Travel Time Functions (TTFs)

To denote that $C = f$ holds for edge $u \rightarrow_{f|C} v$, we usually write $u \rightarrow_f v$ instead. As a periodic p.w.l. function, f is represented as a sequence of *bend points*

$$\langle (x_0, y_0), (x_1, y_1), \ldots, (x_{n-1}, y_{n-1}) \rangle \tag{3.1}$$

with $y_0, y_1, \ldots, y_{n-1} \in \mathbb{R}_{\geq 0}$ and $0 \leq x_0 < x_1 < \cdots < x_{n-1} \leq \Pi$ where Π is the period. Fulfilling the FIFO property is equivalent, in the context of continuous p.w.l. TTFs, to the condition that all line segments have slope -1 or greater. As an example, Figure 3.1 shows two simple periodic continuous p.w.l. TTFs, one of them fulfilling the FIFO property and the other one violating it.

To denote the *complexity* of f—that is, the number of bend points that f has—we write $|f| := n$. Note that this is the same as the number of line segments. The periodicity of TTFs allows us to write (x_i, y_i) even for $i < 0$ and $i \geq n$. To do so, we define

$$(x_i, y_i) := \left(x_{i \bmod n} + \lfloor i/n \rfloor \cdot \Pi,\ y_{i \bmod n} \right)$$

where $i \bmod n$ is the unique number $r \in \{0, 1, \ldots, n-1\}$ with $\exists z \in \mathbb{Z} : x = zn + r$. The value $f(\tau)$ is simply defined as y_i for $\tau = x_i$ with $i \in \mathbb{Z}$. Otherwise, $f(\tau)$ is obtained by linear interpolation. More precisely, for $i_0 \in \mathbb{Z}$ with $\tau \in (x_{i_0}, x_{i_0+1})$ we define

$$f(\tau) := \frac{y_{i_0+1} - y_{i_0}}{x_{i_0+1} - x_{i_0}} \cdot (\tau - x_{i_0}) + y_{i_0} \ . \tag{3.2}$$

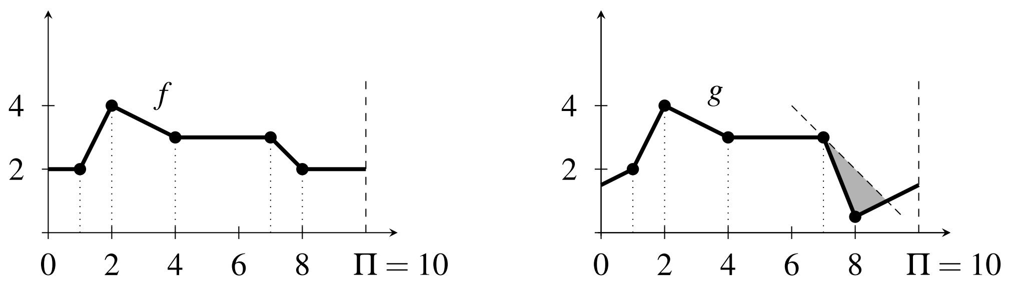


Figure 3.1. The drawing on the left shows a continuous p.w.l. TTF f with the five bend points $(1,2),(2,4),(4,3),(7,3),(8,2)$ and period $\Pi = 10$. The drawing on the right shows the TTF g, a slightly modified version of f where the fifth bend point $(8,2)$ is replaced by $(8,0.5)$. The TTF f fulfills the FIFO property because the smallest occurring slope is -1 (between the bend point $(7,3)$ and $(8,2)$). The TTF g does not fulfill the FIFO property because the slope of the line segment between $(7,3)$ and $(8,0.5)$ is $-2.5 < -1$. The oblique dashed line on the right, which has slope -1, illustrates how g violates the FIFO property.

We sometimes write f_i for the i-th bend point (x_i, y_i). By $f_i.x$ and $f_i.y$ we denote x_i and y_i respectively. To represent a constant TTF—that is, a TTF f with $f \equiv c \in \mathbb{R}_{\geq 0}$—we do not need a sequence of bend points. The number c is already enough. Nevertheless, we define $|f| := 1$ in this case. Note that we generally identify every real number $c \in \mathbb{R}_{\geq 0}$ with the corresponding constant TTF $\tau \mapsto c$. To denote the set of continuous p.w.l. linear functions $f : \mathbb{R} \to \mathbb{R}_{\geq 0}$ that have period Π and fulfill the FIFO property, we write $\mathscr{F}_\Pi$.

Every TTF f implicitly defines an *arrival time function* $\mathbf{arr}\, f := f + \mathrm{id}$; that is, $\mathbf{arr}\, f(\tau) = f(\tau) + \tau$. Obviously, $\mathbf{arr}\, f(\tau)$ is the time one arrives when departing at time τ if traveling takes $f(\tau)$ time. As an example, Figure 3.2 shows a simple periodic continuous p.w.l. TTF and its corresponding arrival time function.

Corollary 3.1. *There is a one-to-one correspondence between a TTF f and its arrival time function* $\mathbf{arr}\, f$. *Moreover, the following three statements hold true:*

1. *f fulfills the FIFO property if, and only if,* $\mathbf{arr}\, f$ *is increasing. That is, $\tau \leq \tau'$ implies* $\mathbf{arr}\, f(\tau) \leq \mathbf{arr}\, f(\tau')$.
2. *f is piecewise linear if, and only if, the same holds true for* $\mathbf{arr}\, f$.
3. *f is continuous if, and only if,* $\mathbf{arr}\, f$ *is continuous. If f is not continuous, then f and* $\mathbf{arr}\, f$ *have the same points of discontinuity.*

The first statement of the above Corollary makes clear how the FIFO property manifests itself in the context of arrival time functions. A TTF $f \in \mathscr{F}_\Pi$ fulfills the FIFO property if, and only if, no line segment of $\mathbf{arr}\, f$ has slope less than 0. This is illustrated in Figure 3.3. Note that the first statement of the above Corollary can

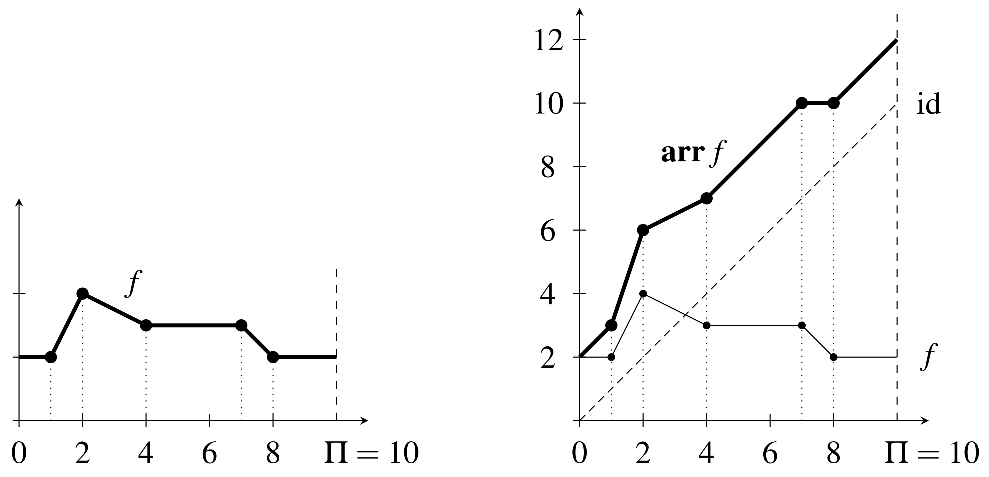


Figure 3.2. The drawing on the left shows TTF f from Figure 3.1. The drawing on the right shows the corresponding arrival time function $\mathbf{arr} f = f + \mathrm{id}$. The dashed line of slope 1 is the graph of the identity function id. The bend points of f correspond to the points $(1,2+1),(2,4+2),(4,3+3),(7,3+7),(8,2+8)$ of $\mathbf{arr} f$. The slope -1, as it occurs in f between the bend points $(7,3)$ and $(8,2)$, corresponds to the slope 0 in $\mathbf{arr} f$.

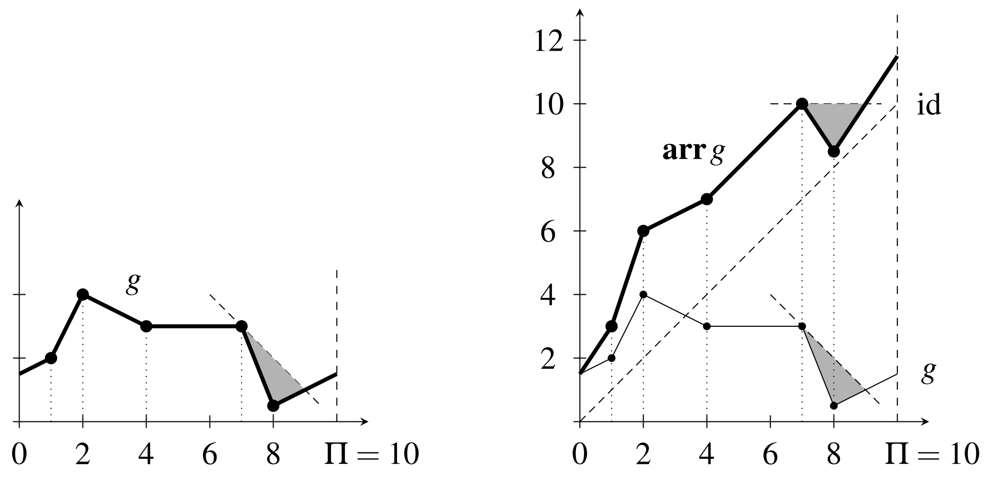


Figure 3.3. The drawing on the left shows TTF g from Figure 3.1, which violates the FIFO property by having slope -2.5 between the bend points $(7,3)$ and $(8,0.5)$. The drawing on the right shows the corresponding arrival time function $\mathbf{arr} g = g + \mathrm{id}$. On the right, one can see what violating the FIFO property means in the context of arrival time functions; namely, that a line segment has a slope less than 0.

further be sharpened in the following way.

Corollary 3.2. *An arrival time function* $\mathbf{arr}\, f$ *is not only increasing but strictly increasing if, and only if, the corresponding TTF* f *fulfills a strengthened version of the FIFO property; that is, if* $\tau < \tau'$ *implies* $f(\tau) + \tau < f(\tau') + \tau$.

To denote the subset of $\mathscr{F}_\Pi$ that only contains TTFs fulfilling the strengthened version of the FIFO property mentioned in Corollary 3.2, we write $\mathscr{F}_\Pi^+$.

We further define the *departure time function* $\mathbf{dep}\, f := (\mathbf{arr}\, f)^{-1}$, which yields the time $\mathbf{dep}\, f(\tau)$ one has departed to arrive at time τ when traveling took a time period of $f(\mathbf{dep}\, f(\tau))$. Of course, $\mathbf{dep}\, f$ is only a real-valued function if $\mathbf{arr}\, f$ is a one-to-one mapping, which means that $\mathbf{arr}\, f$ is strictly increasing (or, equivalently, $f \in \mathscr{F}_\Pi^+$). Otherwise, $\mathbf{dep}\, f(\tau)$ yields a set of possible departure times. The statement

$$\mathbf{dep}\, f \circ \mathbf{arr}\, f = \mathbf{arr}\, f \circ \mathbf{dep}\, f = \mathrm{id} \tag{3.3}$$

follows directly from the definitions of arrival and departure time functions, provided that $f \in \mathscr{F}_\Pi^+$ holds.

3.1.2 Basic Operations on TTFs

For TTFs in $\mathscr{F}_\Pi$ we need three basic operations: *evaluation*, *linking*, and *minimum*. They are explained in the following.

Evaluation of TTFs. Given a TTF $f \in \mathscr{F}_\Pi$ and a departure time $\tau \in \mathbb{R}$ we want to compute the travel time $f(\tau)$. This can be done according to Equation (3.2). However, only the bend points $(x_0, y_0), \ldots, (x_{|f|-1}, y_{|f|-1})$ are really available, but because of the periodicity we have $f(\tau) = f(\tau \bmod \Pi)$.[1] So, we have to find $i_0 \in \{0, \ldots, |f| - 1\}$ such that $\tau \bmod \Pi \in [x_{i_0}, x_{i_0+1})$ holds. With binary search[2], we could do this in $O(\log |f|)$ time. However, we do something different, which we explain later (see Section 3.1.4).

Linking of TTFs. Given two adjacent edges $u \to_f v$ and $v \to_g w$, with $f, g \in \mathscr{F}_\Pi$, we want to know the TTF of the whole path $\langle u \to_f v \to_g f \rangle$. Traveling along the edge $u \to_f v$ takes $f(\tau)$ time when departing at time τ. We hence arrive at the node v at time $\mathbf{arr}\, f(\tau) = f(\tau) + \tau$, which is just the time we depart at node v to travel along the edge $v \to_g w$. So, if we start traveling along the path $\langle u \to_f v \to_g w \rangle$ at

[1]Note that the modulo operation can be extended to real numbers by defining $x \bmod q$ as the unique real number $r \in [0, q)$ with $\exists z \in \mathbb{Z} : x = zq + r$.

[2]Binary search is a well-known method to lookup an element in a sorted sequence in logarithmic time. An explanation can be found in the textbook by Mehlhorn and Sanders for example [61].

time τ, we need $g(f(\tau)+\tau)+f(\tau)$ time to reach w. Accordingly, we define the *linking* of the TTFs f and g as

$$g * f := g \circ \mathbf{arr}\, f + f \tag{3.4}$$

(say g "after" f) to denote the overall TTF of the path $\langle u \rightarrow_f v \rightarrow_g w\rangle$. Linking of TTFs fulfills the properties

$$\mathbf{arr}(g * f) = \mathbf{arr}\, g \circ \mathbf{arr}\, f \tag{3.5}$$
$$\mathbf{dep}(g * f) = \mathbf{dep}\, f \circ \mathbf{dep}\, g \tag{3.6}$$
$$h * (g * f) = (h * g) * f \tag{3.7}$$

for TTFs f, g, h. If $\mathbf{arr}\, f$ or $\mathbf{arr}\, g$ is no one-to-one mapping, then Equation (3.6) speaks about sets instead of real numbers (see Equation (2.10)). To understand Equation (3.5) consider

$$\begin{aligned}\mathbf{arr}\, g \circ \mathbf{arr}\, f &= (g + \mathrm{id}) \circ \mathbf{arr}\, f = g \circ \mathbf{arr}\, f + \mathrm{id} \circ \mathbf{arr}\, f = g \circ \mathbf{arr}\, f + \mathbf{arr}\, f \\ &= g \circ \mathbf{arr}\, f + f + \mathrm{id} = g * f + \mathrm{id} = \mathbf{arr}(g * f)\ .\end{aligned}$$

Equation (3.6) follows easily with Equation (2.11):

$$\begin{aligned}\mathbf{dep}(g * f) &= (\mathbf{arr}(g * f))^{-1} = (\mathbf{arr}\, g \circ \mathbf{arr}\, f)^{-1} \\ &= (\mathbf{arr}\, f)^{-1} \circ (\mathbf{arr}\, g)^{-1} = \mathbf{dep}\, f \circ \mathbf{dep}\, g\end{aligned}$$

A very important property of the linking operation is its associativity as stated by Equation (3.7). To prove this property we calculate

$$h * (g * f) = h \circ \mathbf{arr}(g * f) + g * f = h \circ \mathbf{arr}\, g \circ \mathbf{arr}\, f + g \circ \mathbf{arr}\, f + f$$

as well as

$$\begin{aligned}(h * g) * f &= (h * g) \circ \mathbf{arr}\, f + f = (h \circ \mathbf{arr}\, g + g) \circ \mathbf{arr}\, f + f \\ &= h \circ \mathbf{arr}\, g \circ \mathbf{arr}\, f + g \circ \mathbf{arr}\, f + f\end{aligned}$$

and are finished. The following lemma states a very important property of linking.

Lemma 3.3. *The set $\mathscr{F}_\Pi$ is closed under linking; that is, $g * f$ is again a continuous p.w.l. TTF with period Π and FIFO property for two such TTFs $f, g \in \mathscr{F}_\Pi$.*

Proof. Foschini et al. [36] show in Lemma 2.1 of their paper that the composition $\mathbf{arr}\, g \circ \mathbf{arr}\, f$ of p.w.l. and increasing arrival time functions $\mathbf{arr}\, g$ and $\mathbf{arr}\, f$ is again p.w.l. and increasing. So, with Corollary 3.1 and Equation (3.5), we find that $g * f$ is p.w.l. with FIFO property if the same holds for the TTFs f, g. It is well-known that the composition of continuous functions is again continuous. So, Corollary 3.1 also yields that $g * f$ is continuous if f and g are.

The period Π is also passed from f and g on to $g * f$. To prove that, we remember that the periodicity of f and g is characterized by the conditions $f(\tau + \Pi) = f(\tau)$ and $g(\tau + \Pi) = g(\tau)$ for all $\tau \in \mathbb{R}$. So, with

$$\begin{aligned} g * f(\tau + \Pi) &= g(f(\tau + \Pi) + \tau + \Pi) + f(\tau + \Pi) \\ &= g(f(\tau) + \tau + \Pi) + f(\tau) = g(f(\tau) + \tau) + f(\tau) \\ &= g * f(\tau) \end{aligned}$$

we know that $g * f$ has period Π, too. $\square$

It is relatively easy to see that $\min g + \min f \leq g * f(\tau) \leq \max g + \max f$ holds for all $\tau \in \mathbb{R}$. By induction, this generalizes to

$$\min f_k + \cdots + \min f_1 \leq f_k * \cdots * f_1(\tau) \leq \max f_k + \cdots + \max f_1 \qquad (3.8)$$

for all $\tau \in \mathbb{R}$ and arbitrary $k \in \mathbb{N}$.

Minimum of TTFs. Consider two paths P and R that both go from u to v, where the TTFs $f \in \mathscr{F}_\Pi$ and $g \in \mathscr{F}_\Pi$ describe how long it takes to travel along the path P and R, respectively, depending on the departure time. We want to know the "common TTF" of P and R; that is, the TTF describing how long it takes to travel from u to v if we are allowed to travel along the path P or R (where we always choose the better path for every departure time of course). The resulting merged TTF is just the pointwise minimum of f and g; namely,

$$\min(f, g) \, .$$

The minimum of TTFs is only a special case of the pointwise minimum of real functions that we already considered in Section 2.1.1. From that, the minimum of TTFs inherits associativity and commutativity; that is,

$$\begin{aligned} \min(h, \min(f, g)) &= \min(\min(h, f), g) \\ \min(f, g) &= \min(g, f) \end{aligned}$$

with TTFs f, g, h (see Equations (2.3) and (2.5)). There are two interesting relations; namely

$$\mathbf{arr} \min(f, g) = \min(\mathbf{arr}\, f, \mathbf{arr}\, g) \qquad (3.9)$$

$$\mathbf{dep} \min(f, g) = \max(\mathbf{dep}\, f, \mathbf{dep}\, g) \qquad (3.10)$$

with TTFs f,g, where $\mathbf{arr}\,f, \mathbf{arr}\,g$ have to be one-to-one mappings in case of Equation (3.10). To understand Equation (3.9) it is enough to remember, that the pointwise sum distributes over the pointwise minimum. For Equation (3.10) we argue geometrically that the graph of an inverse function is obtained by just reflecting it at the graph of id as axis. More precisely, for strictly increasing functions $a,b : \mathbb{R} \to \mathbb{R}_{\geq 0}$ we have $\min(a,b)^{-1} = \max(a^{-1}, b^{-1})$ and calculate

$$\begin{aligned}\mathbf{dep}\min(f,g) &= \big(\mathbf{arr}\min(f,g)\big)^{-1} = \big(\min(\mathbf{arr}\,f,\, \mathbf{arr}\,g)\big)^{-1} \\ &= \max\big((\mathbf{arr}\,f)^{-1},\, (\mathbf{arr}\,g)^{-1}\big) = \max(\mathbf{dep}\,f,\, \mathbf{dep}\,g)\ .\end{aligned}$$

If $\mathbf{arr}\,f$ or $\mathbf{arr}\,g$ is no one-to-one mapping, only a weakened version of Equation (3.10) holds true; namely $\mathbf{dep}\min(f,g) \ni \max(\mathbf{dep}\,f \cup \mathbf{dep}\,g)$.

It is also important to note that the set $\mathscr{F}_\Pi$ is closed under minimum, just like in case of linking.

Corollary 3.4. *The set $\mathscr{F}_\Pi$ is closed under minimum; that is,* $\min(f,g)$ *is again a continuous p.w.l. TTF with period* Π *and FIFO property for two such TTFs $f,g \in \mathscr{F}_\Pi$.*

Equivalence of TTFs and Arrival Time Functions. We have learned that the set $\mathscr{F}_\Pi$ is closed under linking and minimum (see Lemma 3.3 and Corollary 3.4). Moreover, with Corollary 3.1 as well as the Equations (3.5) and (3.9), we see that $\mathscr{F}_\Pi$ and $\mathbf{arr}\,\mathscr{F}_\Pi = \{\mathbf{arr}\,f | f \in \mathscr{F}_\Pi\}$ form equivalent structures. More precisely, the one-to-one mapping $f \mapsto \mathbf{arr}\,f$ preserves the structure of linking and minimum and can thus be viewed as an isomorphism[3]. FIFO property and continuity (or the points of discontinuity) are also preserved. But note that the FIFO property is defined differently in the context of arrival time functions: An arrival time function fulfills the FIFO property if it is increasing.

Distributive Property. From the right-distributivity of function composition over minimum (see Equation (2.7)) we inherit the right-distributivity of linking over minimum. More precisely, we have $\min(f,g) * h = \min(f * h,\, g * h)$ for TTFs f,g,h. But interestingly, left-distributivity is also provided. On the first glance, this looks surprising, because linking of TTFs inherits some behavior from the composition of arrival time functions (see Corollary 3.1 and Equation (3.5)), and function composition does not distribute over pointwise minimum in general. But it turns out that left-distributivity follows from the FIFO property.

[3]An *isomorphism* is a structure preserving one-to-one mapping. Here, preserving the structure means that $\mathbf{arr}(f * g) = \mathbf{arr}\,f \circ \mathbf{arr}\,g$ and $\mathbf{arr}\min(f,g) = \min(\mathbf{arr}\,f, \mathbf{arr}\,g)$ holds true.

Lemma 3.5. *With FIFO property, linking of TTFs distributes over minimum, both from the left and the right. That is, for TTFs $f,g,h \in \mathscr{F}_\Pi$ we have:*

$$\min(f,g) * h = \min(f*h,\, g*h) \tag{3.11}$$

$$h * \min(f,g) = \min(h*f,\, h*g) \tag{3.12}$$

Proof. We utilize the aforementioned fact that $f \mapsto \mathbf{arr}\, f$ is an isomorphism and argue completely in terms of arrival time functions, which are equivalent to TTFs. This way, Equation (3.11) follows from Equation (2.7) because of

$$\min(\mathbf{arr}\, f,\, \mathbf{arr}\, g) \circ \mathbf{arr}\, h = \min\big(\mathbf{arr}\, f \circ \mathbf{arr}\, h,\, \mathbf{arr}\, g \circ \mathbf{arr}\, h\big)\ .$$

To prove Equation (3.12), we remember that $\mathbf{arr}\, h$ is increasing because h fulfills the FIFO property. This means $\mathbf{arr}\, h(\mathbf{arr}\, f(\tau)) \le \mathbf{arr}\, h(\mathbf{arr}\, g(\tau))$ holds for $\mathbf{arr}\, f(\tau) \le \mathbf{arr}\, g(\tau)$, which we w.l.o.g. presume. So, we calculate

$$\begin{aligned}\mathbf{arr}\, h \circ \min(\mathbf{arr}\, f,\, \mathbf{arr}\, g)(\tau) &= \mathbf{arr}\, h\big(\min\big\{\mathbf{arr}\, f(\tau),\, \mathbf{arr}\, g(\tau)\big\}\big)\\ &= \min\big\{\mathbf{arr}\, h\big(\mathbf{arr}\, f(\tau)\big),\, \mathbf{arr}\, h\big(\mathbf{arr}\, g(\tau)\big)\big\}\\ &= \min\big(\mathbf{arr}\, h \circ \mathbf{arr}\, f,\, \mathbf{arr}\, h \circ \mathbf{arr}\, g\big)(\tau)\end{aligned}$$

and are finished. □

Calculating with TTFs. Besides the properties of linking and minimum considered so far, there are several more useful calculation rules for TTFs. Consider the TTFs $f,g,h \in \mathscr{F}_\Pi$. Then we have

$$\forall \tau \in \mathbb{R} : \Big(g(\tau) \ge h(\tau) \implies f*g(\tau) \ge f*h(\tau)\Big) \tag{3.13}$$

as well as the somewhat dual statement

$$\forall \tau \in \mathbb{R} : g(\tau) \ge h(\tau) \implies \forall \tau \in \mathbb{R} : g*f(\tau) \ge h*f(\tau)\ . \tag{3.14}$$

Note that Equation (3.14) makes much greater demands on the TTFs g,h than Equation (3.13). To prove Equation (3.13), consider the equivalence

$$\begin{aligned}f*g(\tau) \ge f*h(\tau) &\Leftrightarrow \mathbf{arr}(f*g)(\tau) \ge \mathbf{arr}(f*h)(\tau)\\ &\Leftrightarrow \mathbf{arr}\, f \circ \mathbf{arr}\, g(\tau) \ge \mathbf{arr}\, f \circ \mathbf{arr}\, h(\tau)\\ &\Leftrightarrow \mathbf{arr}\, f\big(g(\tau)+\tau\big) \ge \mathbf{arr}\, f\big(h(\tau)+\tau\big)\ ,\end{aligned}$$

whose last statement is true because of $g(\tau) \ge h(\tau)$ and because f fulfills the FIFO property, which implies $\mathbf{arr}\, f$ is increasing. To prove Equation (3.14), consider the equivalence

$$\begin{aligned}g*f(\tau) \ge h*f(\tau) &\Leftrightarrow \mathbf{arr}\, g \circ \mathbf{arr}\, f(\tau) \ge \mathbf{arr}\, h \circ \mathbf{arr}\, f(\tau)\\ &\Leftrightarrow \mathbf{arr}\, g\big(\mathbf{arr}\, f(\tau)\big) \ge \mathbf{arr}\, h\big(\mathbf{arr}\, f(\tau)\big)\ ,\end{aligned}$$

where the last statement is true, because $g(\tau) \geq h(\tau)$ holds for all $\tau \in \mathbb{R}$. Note that the FIFO property is not needed to prove Equation (3.14), which is contrary to Equation (3.13). There are two important special cases of Equation (3.13) and (3.14):

$$\forall \tau \in \mathbb{R}: \; f * g(\tau) \geq f(\tau) \tag{3.15}$$

$$\forall \tau \in \mathbb{R}: \; f * g(\tau) \geq g(\tau) \tag{3.16}$$

Equation (3.15) is derived from Equation (3.13) by setting $h :\equiv 0$ and calculating $f * g(\tau) \geq f * 0(\tau) = f(0+\tau) + 0 = f(\tau)$. Equation (3.16) is derived from Equation (3.14) by exploiting $f(\tau) \geq 0$ for all $\tau \in \mathbb{R}$ and calculating $f * g(\tau) \geq 0 * g(\tau) = g(\tau)$.

Another interesting question is, what happens if a TTF $f \in \mathscr{F}_\Pi$ is linked with a constant $\gamma \in \mathbb{R}_{\geq 0}$. Linking the constant from the left, we obtain

$$\gamma * f = \gamma + f \; , \tag{3.17}$$

because of $\gamma * f(\tau) = \gamma(f(\tau) + \tau) + f(\tau) = \gamma + f(\tau)$. In other words, f is simply shifted upward by γ. Linking the constant from the right, we obtain

$$f * \gamma(\tau) = \gamma + f(\tau + \gamma) \; , \tag{3.18}$$

which means that f is not only shifted upward, but also to the left, each by an amount of γ.

We also take a closer look at $\mathbf{dep}\, f$ for $f \in \mathscr{F}_\Pi$, including the case that f is no one-to-one mapping. We already pointed out that the “graph” of $\mathbf{dep}\, f$ is obtained by just reflecting the graph of $\mathbf{arr}\, f$ at the graph of id. So, $\min \mathbf{dep}\, f$ and $\max \mathbf{dep}\, f$ are increasing because $\mathbf{arr}\, f$ is increasing, which follows from the fact that f fulfills the FIFO property. Note that the “graph” of $\mathbf{dep}\, f$ never lies above the graph of id for any departure time; that is,

$$\forall \tau \in \mathbb{R}: \;\; \max \mathbf{dep}\, f(\tau) \leq \tau \; . \tag{3.19}$$

This is because $f(\tau) \geq 0$ for all $\tau \in \mathbb{R}$ and this implies that the graph of $\mathbf{arr}\, f$ never lies below the graph of id. The dual statement for arrival function is

$$\forall \tau \in \mathbb{R}: \;\; \mathbf{arr}\, f(\tau) \geq \tau \; . \tag{3.20}$$

A direct consequence of Equation (3.19) is

$$\sigma \in \mathbf{dep}\, f(\tau) \implies \sigma \leq \tau \; . \tag{3.21}$$

3.1.3 Minimum Travel Time Paths

Consider a path $P := \langle u_1 \rightarrow_{f_1} u_2 \rightarrow_{f_2} \cdots \rightarrow_{f_{k-1}} u_k \rangle$ in G with $f_1, \dots, f_{k-1} \in \mathscr{F}_\Pi$. The time needed for traveling from v_1 to v_k along the path P depends on the departure time τ_0 and amounts to $f_P(\tau_0)$ with

$$f_P := f_{k-1} * f_{k-2} * \cdots * f_1 \ . \tag{3.22}$$

The corresponding arrival time at v_k is $f_P(\tau_0) + \tau_0 = \mathbf{arr}\, f_P(\tau_0) = \mathbf{arr}\, f_{k-1} \circ \cdots \circ \mathbf{arr}\, f_1(\tau_0)$ (see Equation (3.5)). We call f_P the *TTF of path P*. Remember that linking is associative and parentheses can be omitted in Equation (3.22) hence. Because of Lemma 3.3 we have $f_P \in \mathscr{F}_\Pi$ and $\mathbf{arr}\, f_P$ is increasing hence.

Lemma 3.6 ([17]). *If every TTF f with $u \rightarrow_f v \in E$ fulfills the FIFO property, then the following two statements hold true:*

1. *Waiting in a node never results in an earlier arrival.*
2. *A cycle in a path never results in an earlier arrival.*

Proof. Dean [17] rightfully points out, that the first statement follows directly from the fact, that $\mathbf{arr}\, f_P$ is increasing for every path P in G. He also points out that every cycle can obviously be replaced by waiting in a node, which implies the second statement. □

Lemma 3.7. *Consider $s, t \in V$ where t is reachable from s in G. Let $\tau_0 \in \mathbb{R}$ be a departure time. Then, there is always a path $P_0 = \langle s \rightarrow \cdots \rightarrow t \rangle$ with minimum travel time. That is, P_0 minimizes $f_P(\tau_0)$ amongst all paths P from s to t.*

Proof. Because of the FIFO property, it is enough to consider only cycle-free paths (due to Lemma 3.6). But there are only finitely many cycle-free paths in a finite directed graph. So, one of these paths, we call it P_0, has minimal arrival time $\mathbf{arr}\, f_{P_0}(\tau_0)$. This directly corresponds to the minimum travel time $f_{P_0}(\tau_0)$. □

Lemma 3.7 ensures that there always exists an *earliest arrival (EA) time* for traveling from a node s to a node t with departure time τ_0 in G; namely, the time

$$\mathrm{EA}_G(s,t,\tau_0) := \min \big\{ f_Q(\tau_0) + \tau_0 \bigm| Q \text{ is path from } s \text{ to } t \text{ in } G \big\} \cup \{\infty\} \ .$$

If $f_P(\tau_0) + \tau_0 = \mathrm{EA}_G(s,t,\tau)$ holds for some path $P = \langle s \rightarrow \cdots \rightarrow t \rangle$, then P is called an *earliest arrival (EA) path* in G. If we want to be more specific, we say that P is an (s,t,τ_0)-EA-path in G. If a node t is not reachable from a node s in G, then we have $\mathrm{EA}_G(s,t,\tau) = \infty$ for all $\tau \in \mathbb{R}$.

Lemma 3.8. *If $\langle u_1 \rightarrow_{f_1} \cdots \rightarrow_{f_{k-1}} u_k \rangle$ is a (u_1, u_k, τ_0)-EA-path in G, then every suffix $\langle u_i \rightarrow_{f_i} \cdots \rightarrow_{f_{k-1}} u_k \rangle$ is a $(u_i, u_k, \mathbf{arr}\, f_{\langle u_1 \rightarrow \cdots \rightarrow u_i \rangle}(\tau_0))$-EA-path in G and we have* $\mathrm{EA}_G(u_1, u_k, \tau_0) = \mathrm{EA}_G(u_i, u_k, \mathbf{arr}\, f_{\langle u_1 \rightarrow \cdots \rightarrow u_i \rangle}(\tau_0)) = \mathrm{EA}_G(u_i, u_k, \mathrm{EA}_G(u_1, u_i, \tau_0))$.

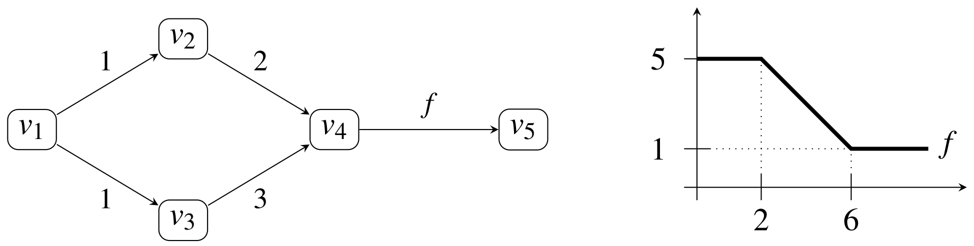


Figure 3.4. Simple time-dependent road network where the travel costs are the travel times. All edges have constant TTFs, except for $v_4 \to_f v_5$. The TTF f, which has slope -1 on the interval $(2,6)$ is depicted on the right. There are two paths from v_1 to v_5 and both of them are $(v_1,v_5,1)$-EA-paths. However, $\langle v_1 \to v_3 \to v_4 \to v_5\rangle$ has a non-optimal prefix. More precisely, $\langle v_1 \to v_3 \to v_4\rangle$ is no $(v_1,v_4,1)$-EA-path, because of $\mathbf{arr}\, f_{\langle v_1\to v_3\to v_4\rangle}(1) = 5 > 4 = \mathbf{arr}\, f_{\langle v_1\to v_2\to v_4\rangle}(1)$.

Proof. With $\mathrm{EA}_G(u_1,u_i,\tau_0) \le \mathbf{arr}\, f_{\langle u_1\to\cdots\to u_i\rangle}(\tau_0)$ and utilizing that all arrival time functions (including $\mathrm{EA}_G(u_i,u_k,\cdot)$) are increasing because of the FIFO property, we calculate

$$\begin{aligned}
\mathrm{EA}_G(u_1,u_k,\tau_0) &= f_{k-1} * \cdots * f_1(\tau_0) + \tau_0\\
&= \mathbf{arr}\, f_{k-1} \circ \cdots \circ \mathbf{arr}\, f_i \circ \mathbf{arr}\, f_{i-1} \circ \cdots \circ \mathbf{arr}\, f_1(\tau_0)\\
&= \mathbf{arr}\, f_{\langle u_i\to\cdots\to u_k\rangle}\big(\mathbf{arr}\, f_{\langle u_1\to\cdots\to u_i\rangle}(\tau_0)\big)\\
&\ge \mathrm{EA}_G\big(u_i,u_k,\mathbf{arr}\, f_{\langle u_1\to\cdots\to u_i\rangle}(\tau_0)\big)\\
&\ge \mathrm{EA}_G\big(u_i,u_k,\mathrm{EA}_G(u_1,u_i,\tau_0)\big)\\
&\ge \mathrm{EA}_G(u_1,u_k,\tau_0)
\end{aligned}$$

and learn that the statement holds true, because there is no earlier arrival time than $\mathrm{EA}_G(u_1,u_k,\tau_0)$ at u_k for departure time τ_0. □

We refer to the property characterized by Lemma 3.8 as *suffix-optimality* of EA paths. The analogous condition for prefix paths does not hold in general. More precisely, a graph G can contain an (s,t,τ_0)-EA-path $\langle s \to \cdots \to u \to \cdots \to t\rangle$ where the prefix $\langle s \to \cdots \to u\rangle$ is not an (s,u,τ_0)-EA-path. Such a situation is only possible if $f_{\langle u\to\cdots\to t\rangle}$ has slope -1 on some interval $[a,b] \subseteq \mathbb{R}$. Then, $\mathbf{arr}\, f_{\langle u\to\cdots\to t\rangle}$ is constant on $[a,b]$ and a later arrival at u does not result in a later arrival at t (provided that $\mathrm{EA}_G(s,u,\tau_0)$ and $f_{\langle s\to\cdots\to u\rangle}(\tau_0)$ both lie in the interval $[a,b]$). Figure 3.4 shows a simple example. However, if no TTF has slope -1, (i.e., $f \in \mathscr{F}_\Pi^+$ for all $u \to_f v \in E$) then all EA paths are prefix-optimal, as Dean already mentions [17].

Corollary 3.9 ([17]). *Let every TTF f with $u \to_f v \in E$ fulfill the strengthened FIFO property mentioned in Corollary 3.2. Then, every prefix $\langle u_1 \to \cdots \to u_i\rangle$ of a (u_1,u_k,τ_0)-EA-path $\langle u_1 \to \cdots \to u_k\rangle$ is a (u_1,u_i,τ_0)-EA-path.*

A slightly weakened version of prefix-optimality, however, is always provided in the context of EA paths, even if the FIFO property is only fulfilled in the non-strict sense.

Lemma 3.10 ([17]). *Consider $s,t \in V$ where t is reachable from s in G. Let $\tau_0 \in \mathbb{R}$ be a departure time. Then, there is always an (s,t,τ_0)-EA-path $\langle s = u_1 \to u_2 \to \cdots \to u_k = t \rangle$ in G such that, for every $i = 1,\ldots,k$, the prefix $\langle s = u_1 \to \cdots \to u_i \rangle$ is an (s,u_i,τ_0)-EA-path. Such an EA path is called* <u>*prefix-optimal.*</u>

Proof. Let $P := \langle s = v_1 \to_{f_1} \cdots \to_{f_{\ell-1}} v_\ell = t \rangle$ be a cycle-free (s,t,τ_0)-EA-path in G. Choose the maximal $i \in \{2,\ldots,\ell\}$ such that the prefix $P_i := \langle s \to \cdots \to v_i \rangle$ of P is not an (s,v_i,τ_0)-EA-path. Lemma 3.7 guaranties the existence of a cycle-free (s,v_i,τ_0)-EA-path R and we get

$$\begin{aligned}
\mathrm{EA}_G(s,t,\tau_0) &= \mathbf{arr}\, f_P(\tau_0) \\
&= \mathbf{arr}\, f_{\langle v_i \to \cdots \to t \rangle} \circ \mathbf{arr}\, f_{P_i}(\tau_0) \\
&\geq \mathbf{arr}\, f_{\langle v_i \to \cdots \to t \rangle} \circ \mathbf{arr}\, f_R(\tau_0) \\
&\geq \mathrm{EA}_G(s,t,\tau_0)\ .
\end{aligned}$$

So, we have $\mathbf{arr}\, f_{R\langle v_i \to \cdots \to t \rangle} = \mathrm{EA}_G(s,t,\tau_0)$ and replacing the prefix P_i of P by R yields another (s,t,τ_0)-EA-path P' hence.

The new cycle-free path $P' = \langle s = w_1 \to \cdots \to w_{\ell'} = t \rangle$ has the following important property: If we choose the maximal $j \in \{2,\ldots,\ell'\}$, such that the prefix $P'_j := \langle s \to \cdots \to w_j \rangle$ of P' is not an (s,w_j,τ_0)-EA-path, then w_j comes on P' before v_i; that is,

$$P' = \langle s \to \cdots \to w_j \to \cdots \to v_i \to \cdots \to t \rangle\ .$$

So, if we repeatedly replace maximal prefix paths in the described manner, this process surely terminates. This is because cycle-free paths can not have arbitrarily many hops in a finite graph like G, and the suffix of the path $\langle w_j \to \cdots \to v_i \to \cdots \to t \rangle$ of P' gains at least one hop in every step. The resulting path is a prefix-optimal (s,t,τ_0)-EA-path after termination. □

Dean also proves Lemma 3.10 by repeatedly replacing subpaths of EA path with EA paths [17]. But his procedure is non-deterministic and he does not explain whether it ever terminates.

A directed tree $T = (V_T, E_T) \subseteq G$ with root s, where each path $P_u \subseteq T$ from s to $u \in V_T$ is an (s,u,τ_0)-EA-path in G, is called an *earliest arrival (EA) tree*. Or if we want to be more specific, we say T is an (s,τ_0)-EA-tree. The unique EA paths in T are all prefix-optimal of course.

Obviously, the minimum time duration for traveling from s to t for departure time $\tau_0 \in \mathbb{R}$ can be obtained from the EA time easily and amounts to

$$\mathrm{TT}_G(s,t,\tau_0) := \mathrm{EA}_G(s,t,\tau_0) - \tau_0 \ .$$

Note that $\mathrm{TT}_G(s,t,\cdot)$ forms a TTF; namely, the TTF that yields the minimum possible travel time from s to t for all departure times. In fact, $\mathrm{TT}_G(s,t,\cdot)$ is exactly what we call a travel time profile (TTP) in Section 1.1.2.

As $\mathrm{TT}_G(s,t,\cdot)$ is the minimum of the TTFs of finitely many cycle-free paths from s to t, we can be sure that $\mathrm{TT}_G(s,t,\cdot) \in \mathscr{F}_\Pi$ holds. Obviously, $\mathrm{EA}_G(s,t,\tau_0)$ and $\mathrm{TT}_G(s,t,\cdot)$ is just what an EA query and a TTP query computes respectively (see Section 1.1.2). Dual to the EA time is the interval of *latest departure (LD) times*

$$\mathrm{LD}_G(s,t,\sigma_0) := \big(\mathbf{dep}\,\mathrm{TT}_G(s,t,\cdot)\big)(\sigma_0) \ \subseteq \ \mathbb{R}$$

for traveling from s to t with arrival time σ_0. The respective kind of one-to-one query that computes $\mathrm{LD}_G(s,t,\sigma_0)$ for given $s,t \in V$ and $\sigma_0 \in \mathbb{R}$ is called an *LD query*. Obviously, $\mathrm{LD}_G(s,t,\cdot) = \mathbf{dep}\,\mathrm{TT}_G(s,t,\cdot) = (\mathrm{EA}_G(s,t,\cdot))^{-1}$ holds. If all TTFs in G fulfill the strengthened FIFO property, then $\mathrm{LD}_G(s,t,\cdot)$ is a real function. We further define an upper and a lower bound of the minimum travel times from s to t; namely,

$$\overline{\mathrm{TT}}_G(s,t) := \min\Big\{\sum_{i=1}^{k-1} \max f_i \ \Big|\ \langle s = u_1 \to_{f_1} \cdots \to_{f_{k-1}} u_k = t\rangle \subseteq G\Big\} \cup \{\infty\} \ ,$$

$$\underline{\mathrm{TT}}_G(s,t) := \min\Big\{\sum_{i=1}^{k-1} \min f_i \ \Big|\ \langle s = u_1 \to_{f_1} \cdots \to_{f_{k-1}} u_k = t\rangle \subseteq G\Big\} \cup \{\infty\} \ .$$

They fulfill $\underline{\mathrm{TT}}_G(s,t) \le \mathrm{TT}_G(s,t,\tau) \le \overline{\mathrm{TT}}_G(s,t)$ for all $\tau \in \mathbb{R}$ thanks to Equation (3.8).

3.1.4 Computing Basic Operations on TTFs Efficiently

In Section 3.1.2 we consider the properties of the three basic operations on TTFs (i.e., evaluation, linking, and minimum). Here, we discuss how the three operations can be computed efficiently. This is interesting, because all three basic operations are invoked as subprocedures by the route planning algorithms described in this thesis and affect the performance of these algorithms hence. We also discuss, how complex the resulting TTFs of linking and minimum can get.

Efficient Evaluation of TTFs. When computing the evaluation $f(\tau_0)$ of a TTF $f \in \mathscr{F}_\Pi$ for an argument $\tau_0 \in \mathbb{R}$, we want to avoid the binary search mentioned earlier (see page 81) to get rid of the logarithmic factor. So, the idea is to partition $[0,\Pi)$ into $\lfloor |f|/4 \rfloor$ intervals, each of length $\Pi/\lfloor |f|/4 \rfloor$. Each such interval is associated with a bucket. The bend points that lie in each interval have to be stored in the associated bucket. If the x-values of the bend points are uniformly distributed, then i with $\tau_0 \in [x_i, x_{i+1})$ is found in $O(1)$ average time by searching the bucket with number

$$b := \Big\lfloor (\tau_0 \bmod \Pi) \cdot \lfloor |f|/4 \rfloor / \Pi \Big\rfloor \tag{3.23}$$

only. We also want to avoid expensive divisions and multiplications during evaluation.[4] So, what we really do is to partition $[0,\Pi)$ into

$$K_f := \left\lfloor \frac{\Pi}{2^{\lfloor \log(4\Pi/|f|) \rfloor}} \right\rfloor$$

buckets of length $L_f := 2^{\lfloor \log(4\Pi/|f|) \rfloor}$, which fulills $|f|/4 - 1 \le K_f \le |f|/2$ and $2\Pi/|f| \le L_f \le 4\Pi/|f|$. This has the advantage that Equation (3.23), which contains division and multiplication, can be replaced by the expression

$$b := \max\Big\{0, \big(\lfloor \tau_0 \bmod \Pi \rfloor \gg \lfloor \log(4\Pi/|f|) \rfloor\big) - 1\Big\}\,, \tag{3.24}$$

which only contains a much cheaper right shift. Note that the number $B_f := \lfloor \log(4\Pi/|f|) \rfloor \in \mathbb{N}_0$ has to be stored together with the bucket and bend point data of a TTF f. This means the logarithm and the division are only computed once; namely, when the data of f is set up.

Some readers may notice that computing $\lfloor \tau_0 \bmod \Pi \rfloor$ also includes division. Within the context of time-dependent route planning, however, it is not likely that τ_0 gets larger than $k \cdot \Pi$ for a small $k \in \mathbb{N}$. For $\tau_0 \in [\Pi, 2\Pi)$, for example, we simply compute $\tau_0 - \Pi$ to obtain $\tau_0 \bmod \Pi$. So, in most cases $\tau_0 \bmod \Pi$ can be computed without division too. The multiplication in Equation (3.2), however, can not be avoided. But the division in Equation (3.2) could be avoided by storing the slopes of all $|f|$ line segments. This would need $\Theta(|f|)$ additional space of course.

Efficient Linking of TTFs. The linking $g * f$ of TTFs $f, g \in \mathscr{F}_\Pi$ can be computed in $O(|g| + |f|)$ time using Algorithm 3.1. On the one hand, this algorithm

[4]Multiplication and especially division still belong to the slower operations provided by todays CPUs. Replacing them by cheaper operations can speedup computations considerably. The integer division of k by 2^ℓ (with $k, \ell \in \mathbb{N}$) is equivalent to shifting the binary representation of k to the right by ℓ digits. We denote this right shift by $k \gg \ell$. So, we have $k \gg \ell = \lfloor k/2^\ell \rfloor$. Right shifts are usually performed very fast.

works similar to the merging of sorted sequences[5], where $\langle g_0.x, \ldots, g_{|g|-1}.x\rangle$ and $\langle f_0.x + f_0.y, \ldots, f_{|f|-1}.x + f_{|f|-1}.y\rangle$ are considered as the sequences to be merged (the second sequence consists actually of the y-values of the bend points of $\mathbf{arr}\, f$). On the other hand, the algorithm can be viewed as a simple sweep line algorithm. The running time obviously lies in $\mathrm{O}(|f| + |g|)$. To understand the algorithm, it is also helpful to consider the equivalent composition of the according arrival time functions; that is, $\mathbf{arr}\, g \circ \mathbf{arr}\, f$. The algorithm computes $\mathbf{arr}\, g \circ \mathbf{arr}\, f$, but stores $g * f = \mathbf{arr}\, g \circ \mathbf{arr}\, f - \mathrm{id}$. That is, it successively appends bend points (x, y) to the sequence *result*, such that $(x, y + x)$ is a bend point of $\mathbf{arr}\, g \circ \mathbf{arr}\, f$.

The composition $\mathbf{arr}\, g \circ \mathbf{arr}\, f$ is actually a transformed version of $\mathbf{arr}\, g$. It is obtained by partly stretching and partly shrinking $\mathbf{arr}\, g$ in x-direction. Which parts of $\mathbf{arr}\, g$ are stretched or shrunken depends on the different slopes of the line segments of $\mathbf{arr}\, f$: shrinking, if the slope lies in the interval $(1, \infty)$, stretching, if the slope lies in the interval $(0, 1)$, and neither shrinking nor stretching if the slope is 1. If the slope of $\mathbf{arr}\, f$ is 0 on an interval $[a, b]$, then $\mathbf{arr}\, g \circ \mathbf{arr}\, f$ is constant on $[a, b]$. For example, if $\mathbf{arr}\, f$ has slope 2 on an interval $[a, b]$, then $[\mathbf{arr}\, f(a), \mathbf{arr}\, f(b)]$ has length $2 \cdot (b - a)$. But this means, that $\mathbf{arr}\, g \circ \mathbf{arr}\, f$ runs through the same values on the interval $[a, b]$ as $\mathbf{arr}\, g$ on the interval $[\mathbf{arr}\, f(a), \mathbf{arr}\, f(b)]$, which has double length. So, $\mathbf{arr}\, g$ appears shrunken by the factor $1/2$ as part of $\mathbf{arr}\, g \circ \mathbf{arr}\, f$ on $[a, b]$.

Note that $\mathbf{arr}\, g$ is not only shrunken or stretched, but also shifted to the left. How far a point of $\mathbf{arr}\, g$ is shifted depends on $\mathbf{arr}\, f$. More precisely, the point $\big(\mathbf{arr}\, f(\tau), \mathbf{arr}\, g(\mathbf{arr}\, f(\tau))\big)$ of $\mathbf{arr}\, g$ is shifted to the left by $f(\tau)$. This means that the corresponding point of g is effectively shifted to the upper left by $f(\tau)$. So, $\mathbf{arr}\, g \circ \mathbf{arr}\, f$ is a left shifted and shrunken or stretched version of $\mathbf{arr}\, g$. Correspondingly, $f * g = \mathbf{arr}\, g \circ \mathbf{arr}\, f - \mathrm{id}$ is a left upward shifted and shrunken or stretched version of g. Figure 3.5 illustrates the linking of two simple TTFs of period 10.

The composition $\mathbf{arr}\, g \circ \mathbf{arr}\, f$ has three kinds of bend points that are treated by the three different cases of the conditional construction inside the loop.

- First, bend points that originate from f; that is, from f_j for some index $j \in \{0, \ldots, |f| - 1\}$. The corresponding bend point of $\mathbf{arr}\, g \circ \mathbf{arr}\, f$ is
$$\big(f_j.x, \mathbf{arr}\, g(\mathbf{arr}\, f(f_j.x))\big) = \big(f_j.x, \mathbf{arr}\, g(f_j.x + f_j.y)\big)\ .$$
Its y-coordinate is computed from the line segment $(g_{i-1} g_i)$ by linear interpolation, as we have $f_j.x + f_j.y \in [g_{i-1}.x,\, g_i.x)$. This kind of bend points is treated in Line 14 to 18 of Algorithm 3.1.

[5]Merging of sorted sequences is an algorithmic standard technique. The well-known sorting algorithm merge sort, for example, is nothing but recursively applying the idea that already sorted subsequences can be merged in linear time by scanning them simultaneously. More details can be found in the textbook by Mehlhorn and Sanders [61] for example.

Algorithm 3.1. Given the TTFs $f, g \in \mathscr{F}_\Pi$, this algorithm computes $g * f$. As TTFs are represented as sequences of bend points with strictly increasing x-values, the resulting TTF is build simply by appending the computed bend points to the end of the sequence *result*.

```
function $linkTTF(g, f : TTF) : TTF$
    $result := \langle\rangle : Sequence$
    $i := \min\{k \,|\, g_k.x \geq f(0)\}$
    $j := 0$
    while $true$ do
        if $g_i.x = f_j.x + f_j.y$ then
            $(x, y) := (f_j.x,\ g_i.y + f_j.y)$            // $g_i.y + f_j.y = g_i.x + g_i.y - f_j.x$
            $i := i + 1,\ j := j + 1$
        else if $g_i.x < f_j.x + f_j.y$ then
            $m_{\mathbf{arr}\,f} := (f_j.x + f_j.y - f_{j-1}.x - f_{j-1}.y)/(f_j.x - f_{j-1}.x)$
            $x := 1/m_{\mathbf{arr}\,f} \cdot (g_i.x - f_{j-1}.x - f_{j-1}.y) + f_{j-1}.x$
            $y := g_i.x + g_i.y - x$
            $i := i + 1$
        else
            $m_g := (g_i.y - g_{i-1}.y)/(g_i.x - g_{i-1}.x)$
            $x := f_j.x$
            $y := g_{i-1}.y + m_g \cdot (f_j.x + f_j.y - g_{i-1}.x) + f_j.y$
            $j := j + 1$
        if $x \geq \Pi$ then break
        if $|result| \geq 2$ then
            let $p, q$ be the last two bend points of $result$
            if $ccw(p, q, (x, y)) = 0$ then remove last bend point from $result$
        append bend point $(x, y)$ to $result$
    return $result$
```

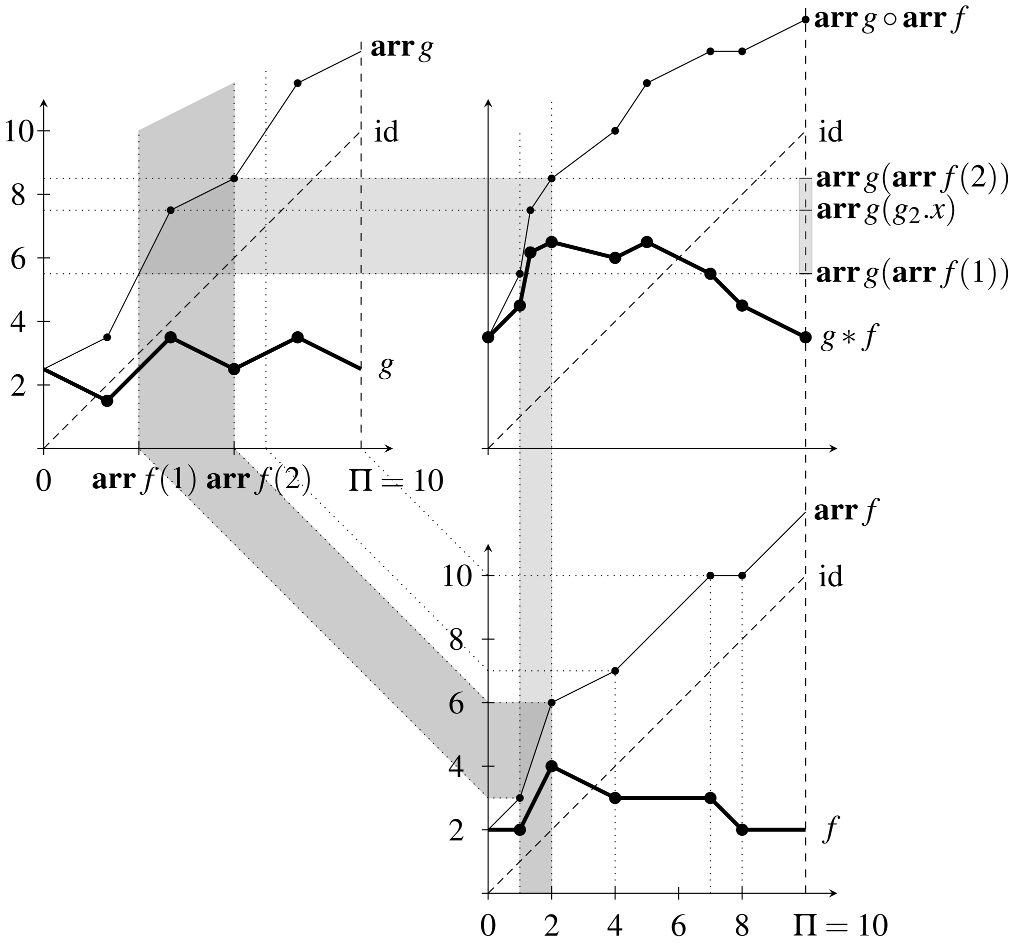


Figure 3.5. Linking $g * f$ of two simple TTFs $f, g \in \mathscr{F}_{10}$. The result $g * f$ is the same as $\mathbf{arr}\, g \circ \mathbf{arr}\, f - \mathrm{id}$. So, in principle, $g * f$ can be obtained by computing $\mathbf{arr}\, g \circ \mathbf{arr}\, f$, which is the same as shrinking and stretching $\mathbf{arr}\, g$ depending on the slopes that occur in $\mathbf{arr}\, f$. On the interval $[1,2]$, for example, $\mathbf{arr}\, f$ has slope 3. So, $\mathbf{arr}\, g|_{[\mathbf{arr}\, f(1), \mathbf{arr}\, f(2)]} = \mathbf{arr}\, g|_{[3,6]}$ appears on interval $[1,2]$ in $\mathbf{arr}\, g \circ \mathbf{arr}\, f$; that is, shrunken by a factor of $1/3$. The corresponding parts of $\mathbf{arr}\, f$, $\mathbf{arr}\, g$, and $\mathbf{arr}\, g \circ \mathbf{arr}\, f$ are highlighted gray.

- Second, bend points that originate from g; that is, from g_i for some index $i \in \mathbb{Z}$. The corresponding bend point of $\mathbf{arr}\, g \circ \mathbf{arr}\, f$ is $(x_f, \mathbf{arr}\, g(g_i.x)) = (x_f, g_i.x + g_i.y)$ for some $x_f \in [f_{j-1}.x, f_j.x)$ with
$$g_i.x \in [f_{j-1}.x + f_{j-1}.y, f_j.x + f_j.y) \ .$$
The x-coordinate x_f is obtained by linear interpolation of the line segment $(f_{j-1} f_j)$ (see Line 9 to 13 of Algorithm 3.1).
- Third, bend points that originate from both f and g; that is, $g_i.x = f_j.x + f_j.y = \mathbf{arr}\, f(f_j.x)$. The corresponding bend point of $\mathbf{arr}\, g \circ \mathbf{arr}\, f$ is simply $(f_j.x, g_i.x + g_i.y)$. This kind of bend points is very easy to deal with, because the x- and the y-coordinate can both be taken directly from g and f, without any linear interpolation (see Line 6 to 8 of Algorithm 3.1).

This makes clear why the linking of TTFs can be done in a way that is similar to the merging of sorted sequences. One simply goes through the intervals $[f_{j-1}.x, f_j.x]$ with $j \in \{0, \dots, |f| - 1\}$ and, for each such interval, through all bend points of g_i with
$$g_i.x \in \big[\mathbf{arr}\, f(f_{j-1}.x), \mathbf{arr}\, f(f_j.x)\big] = \big[f_{j-1}.x + f_{j-1}.y, f_j.x + f_j.y\big] \ .$$
The three possible cases just discussed also make clear how complex the linked TTF $g * f$ can get. A bend point of $g * f = \mathbf{arr}\, g \circ \mathbf{arr}\, f - \mathrm{id}$ can only emerge from a bend point of g, a bend point of f, or of both—a consideration also made by Foschini et al. (see Lemma 2.4 of their paper [36]).

Corollary 3.11. *Consider $f, g \in \mathscr{F}_\Pi$. Then, we have $|g * f| \leq |g| + |f|$ and this bound is tight.*

There are two reasons why $|g * f|$ can be smaller than $|f| + |g|$. First, if $g_i.x = f_j.x + f_j.y$ occurs, then we append one bend point that corresponds to two original bend points (see Line 6 to 8). Second, if we append a bend point (x, y) to *result* $=: \langle p_1, \dots, p_k \rangle$, such that $\langle p_{k-1}, p_k, (x, y) \rangle$ are collinear, then p_k can be removed (see Line 20 to 22).

Efficient Minimum of TTFs. The minimum $\min(f, g)$ of TTFs $f, g \in \mathscr{F}_\Pi$ can be computed with Algorithm 3.2. This algorithm is, like in case of linking, a simple sweep line algorithm. It is also similar to merging two sorted sequences, where the sequences are the x-values of the bent points of f and g; that is, the sequences $\langle f_0.x, \dots, f_{|f|-1}.x \rangle$ and $\langle g_0.x, \dots, g_{|g|-1}.x \rangle$. Again, the running time lies in $\mathrm{O}(|f| + |g|)$.

While the algorithm runs, it successively appends points to the sequence *result*. There are four kinds of bend points of $\min(f, g)$. They are treated by the conditional construction inside the loop.

Algorithm 3.2. Given the continuous piecewise linear TTFs f,g with period Π, this algorithm computes $\min(f,g)$. Like in Algorithm 3.1, the computed bend points are successively appended to the end of the sequence *result*.

```
function $minTTF(f,g : TTF) : TTF$
    $result := \langle\rangle : Sequence$
    $i := j := 0$
    while $j < |f|$ or $i < |g|$ do
        if $|(f_{j-1}f_j) \cap (g_{i-1}g_i)| = 1$ then
            let $(x,y)$ be the intersection point of $(f_{j-1}f_j)$ and $(g_{i-1}g_i)$
            if $x \geq 0$ then append bend point $(x,y)$ to $result$
        if $f_j.x = g_i.x$ then
            if $f_j.y = g_i.y$ then append bend point $(f_j.x, f_j.y)$ to $result$
            else append bend point $(f_j.x, \min\{f_j.y, g_i.y\})$ to $result$
            $j := j+1, i := i+1$
        else if $f_j.x < g_i.x$ then
            if $ccw(g_{i-1}, f_j, g_i) \leq 0$ then
                if $ccw(g_{i-1}, f_j, g_i) = 0$ then
                    if $ccw(g_{i-1}, f_{j-1}, g_i) < 0$ or $ccw(g_{i-1}, f_{j+1}, g_i) < 0$ then
                        append bend point $f_j$ to $result$
                else append bend point $f_j$ to $result$
            $j := j+1$
        else
            if $ccw(f_{j-1}, g_i, f_j) \leq 0$ then
                if $ccw(f_{j-1}, g_i, f_j) = 0$ then
                    if $ccw(f_{j-1}, g_{i-1}, f_j) < 0$ or $ccw(f_{j-1}, g_{i+1}, f_j) < 0$ then
                        append bend point $g_i$ to $result$
                else append bend point $g_i$ to $result$
            $i := i+1$
    if $|(f_{|f|-1}f_{|f|}) \cap (g_{|g|-1}g_{|g|})| = 1$ then
        let $(x,y)$ be the intersection point of $(f_{|f|-1}f_{|f|})$ and $(g_{|g|-1}g_{|g|})$
        if $x < \Pi$ then append bend point $(x,y)$ to $result$
    return $result$
```

- First, there are the bend points that emerge from the input TTF f or g if one of the TTFs lies beneath the other. To find out whether f or g lies beneath, we often use the predicate *ccw* explained in Section 2.1.4. This kind of bend points is appended in Line 10, 17, and 24.
- Second, there are bend points that emerge from the input TTF f or g if a bend point lies on a line segment of the other TTF each. Such a bend point is not always appended. A bend point of f_j lying on a segment $(g_{i-1}g_i)$ is added if the segment $(f_{j-1}f_j)$ or (f_jf_{j+1}) lies beneath $(g_{i-1}g_i)$ (see Line 14 to 16). Appending a bend point g_j lying on a segment $(f_{j-1}f_j)$ depends on the analogous condition (see Line 21 to 23).
- Third, there are the bend points that emerge both from the input TTFs f and g. More precisely, if f and g have a bend point in common, then this point is appended to *result*. This kind of bend points is appended in Line 9.
- Fourth, there are bend points that emerge from an intersection of f and g. Such points have to appear in the result TTF because they are the points where it changes whether f or g is beneath. This kind of bend points is appended in Line 7 and 28.

Considering Algorithm 3.2 makes clear how complex a minimum TTF $\min(f,g)$ can get. The loop is executed not more than $|f|+|g|$ times never appending more than two bend points each. Having finished the loop, one additional bend point may be appended.

Corollary 3.12. *Consider $f,g \in \mathscr{F}_\Pi$. Then, we have $|\min(f,g)| \le 2|f|+2|g|+1$.*

That $|\min(f,g)|$ is smaller than $2|f|+2|g|+1$ is not unlikely. If f and g only intersect rarely, then several bend points of f and g do not appear in $\min(f,g)$ while few additional bend points (i.e., intersection points) are added.

3.2 Time-Dependency with Additional Costs

We now consider the more general case that the travel cost is different from the travel time; that is, $C \neq f$ for $u \rightarrow_{f|C} v \in E$ in general. However, the TCFs associated with the edges of G are still relatively restricted in this thesis; that is, $C(\tau) = f(\tau) + c$ with $c \in \mathbb{R}_{\ge 0}$ only depending the edge and not on the departure time. This is, as already said, where the term of "additional time-invariant costs" comes from. We usually write $u \rightarrow_{f|c} v$ instead of $u \rightarrow_{f|f+c} v$.

This kind of TCFs can be used to model travel costs where traveling an edge is proportional to travel time with an additional time-invariant offset. An example is the computation of routes that minimize the salary of a driver, which we assume to be proportional to travel time, together with toll costs occurring on motorways, which are assumed to be time-invariant. Another example is the computation of routes with small travel time, but inconvenient roads (e.g., narrow winding, steep, bumpy) are avoided by making them more expensive with an additional penalty.

If $c = 0$ holds for all edges $u \to_{f|c} v$ in G, then we have the special case that the travel cost is simply the travel time; that is, $C = f$. If $f \in \mathscr{F}_\Pi$ holds, then we also have $C = f + c \in \mathscr{F}_\Pi$. It is very important to note that the travel time is part of the travel cost in this setup. So, waiting is not for free, but waiting an amount of time $\Delta\tau$ exactly raises the cost $\Delta\tau$.

First in this section, we consider the properties of minimum cost paths for time-dependent travel times with additional constant (i.e., time-invariant) costs (see Section 3.2.1). Then, we extend the basic operations on TFFs to work with additional time-invariant costs. However, TCFs $f + c$ with a TTF $f \in \mathscr{F}_\Pi$ and an additional constant cost $c \in \mathbb{R}_{\geq 0}$ are not closed under the minimum operation. To fix that, we allow f to have points of discontinuity and c to be a piecewise constant function instead of a simple constant value. Now, c describes an additional time-*dependent* cost (see Section 3.2.2). We then explain how the resulting generalized basic operations can be computed efficiently (see Section 3.2.3). That the generalized basic operations not only include time-invariant but also piecewise constant (i.e., time-dependent) costs, raises the question how minimum cost paths behave in such a more general setup (see Section 3.2.4).

3.2.1 Minimum Cost (MC) Paths

Consider a path $P := \langle u_1 \to_{f_1|c_1} u_2 \to_{f_2|c_2} \cdots \to_{f_{k-1}|c_{k-1}} u_k \rangle$ in G with $f_i \in \mathscr{F}_\Pi$ and $c_i \in \mathbb{R}_{\geq 0}$ for $i = 1, \ldots, k-1$. The time needed for traveling from v_1 to v_k along the path P for departure time τ_0 is $f_P(\tau_0) = f_{k-1} * f_{k-2} * \cdots * f_1(\tau_0)$ (see Equation (3.22)). The total *additional time-invariant cost* along P is independent of the departure time τ_0 and amounts to

$$c_P := c_1 + c_2 + \cdots + c_{k-1} \ . \tag{3.25}$$

So, the total time-dependent cost of traveling along P when departing at τ_0 is simply the sum

$$C_P(\tau_0) := f_P(\tau_0) + c_P \ . \tag{3.26}$$

Of course, C_P is a TCF; namely, the *TCF of path P*. With $f_P \in \mathscr{F}_\Pi$ and $c_P \in \mathbb{R}_{\geq 0}$, we obviously have $C_P \in \mathscr{F}_\Pi$. That is, C_P is a periodic p.w.l. function formally fulfilling the FIFO property. We define an upper and a lower bound of C_P; namely, $\overline{C}_P := \max f_1 + \cdots + \max f_{k-1} + c_P$ and $\underline{C}_P := \min f_1 + \cdots + \min f_{k-1} + c_P$ respectively.

Like in the special case, where travel times and travel costs are the same, cycles are not beneficial.

Lemma 3.13. *If every TTF f with $u \rightarrow_{f|c} v \in E$ fulfills the FIFO property, then cycles in paths never result in a lower travel cost.*

Proof. From Lemma 3.6 we know that cycles never result in lower travel times. The additional time-invariant costs do not depend on time, which means that cycles can only increase the total additional time-invariant costs. So, cycles can only increase time-dependent travel costs. □

Remember that travel cost is travel time plus a time-invariant additional cost. Especially, travel time is part of the travel cost, which means that we do not consider waiting time as free. So, waiting is, just like cycles, not beneficial either.

Lemma 3.14. *If every TTF f with $u \rightarrow_{f|c} v \in E$ fulfills the FIFO property, then waiting in a node never result in a lower travel cost.*

Proof. From Lemma 3.13 we know that cycles never results in lower travel costs. But waiting an amount of time $\Delta\tau$ in a node u can always be simulated by adding a cycle $\langle u \rightarrow_{h|0} u' \rightarrow_{h'|0} u \rangle$ with $h \equiv h' \equiv \Delta\tau/2$. So, waiting is not beneficial. □

Lemma 3.15. *Consider $s,t \in V$ where t is reachable from s in G. Let $\tau_0 \in \mathbb{R}$ be a departure time. Then, there is always a path $P_0 = \langle s \rightarrow \cdots \rightarrow t \rangle$ with minimum travel cost. That is, P_0 minimizes $C_P(\tau_0)$ amongst all paths P from s to t.*

Proof. We argue as in the proof of Lemma 3.7: From Lemma 3.13 we know cycles can be ignored. But there are only finitely many cycle-free paths from s to t in G. At least one of them has minimum total travel cost for departure time τ_0. □

Lemma 3.15 ensures that there always exists a *minimum travel cost* from a node s to a node t for departure time τ_0; namely, the cost

$$\operatorname{Cost}_G(s,t,\tau_0) := \min \left\{ C_Q(\tau_0) \;\middle|\; Q \text{ is path from } s \text{ to } t \text{ in } G \right\} \cup \{\infty\} . \tag{3.27}$$

If $P = \langle s \rightarrow \cdots \rightarrow t \rangle$ fulfills $C_P(\tau_0) = \operatorname{Cost}(s,t,\tau_0)$, then P is called a *minimum cost (MC)* path in G. To be more specific, we say that P is an (s,t,τ_0)-MC-path.

In Section 3.1.3 we learned, that there is always a prefix-optimal (s,t,τ_0)-EA path for all $s,t \in V$ and $\tau_0 \in \mathbb{R}$ (provided that t is reachable from s). However, things are much less convenient in the presence of additional time-invariant costs because the existence of prefix-optimal MC paths is no longer guaranteed. More precisely, it can happen that all MC paths for given $s,t \in V, \tau_0 \in \mathbb{R}$ have a prefix path that is not an MC path. Figure 3.6 shows a simple example of such a situation. This lack of *prefix-optimality* implies that Dijkstra-like algorithms do not yield the correct result if they only maintain a single label per node. This is because maintaining only one label per node means that suboptimal intermediate results

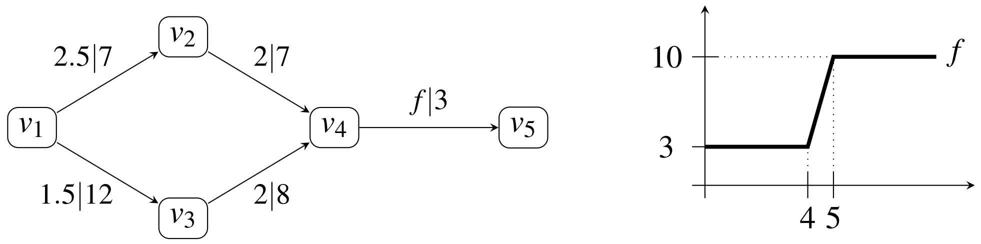


Figure 3.6. A simple time-dependent example graph with additional time-invariant costs. Most edges have constant TTFs. Only $v_4 \to v_5$ has the non-constant TTF f depicted on the right. The path $\langle v_1 \to v_3 \to v_4 \to v_5 \rangle$ is the only minimum total cost path from v_1 to v_5 for departure time 0.5, with $\text{Cost}(v_1, v_5, 0.5) = 6.5 + 23 = 29.5$ (a travel time of 6.5 and a total additional cost of 23). Its prefix path $\langle v_1 \to v_3 \to v_4 \rangle$, however, has total cost $23.5 = 3.5 + 20$ and is not a $(v_1, v_4, 0.5)$-MC-path, because $\langle v_1 \to v_2 \to v_4 \rangle$ has a smaller total cost of $18.5 = 4.5 + 14$. The path $\langle v_1 \to v_2 \to v_4 \to v_5 \rangle$, which has total cost $31.5 = 14.5 + 17$ for departure time 0.5, is not a $(v_1, v_5, 0.5)$-MC-path of course. The arrival times at node v_4 are the travel times plus the departure time 0.5. So, in case of $\langle v_1 \to v_3 \to v_4 \rangle$ the arrival time is 4 and the TTF f yields $f(4) = 3$, which makes $\langle v_1 \to v_3 \to v_4 \to v_5 \rangle$ a $(v_1, v_5, 0.5)$-MC-path.

are thrown away. Instead, one has to perform time-dependent multi-label search (see Section 4.4).

So, there is no guarantied prefix-optimality in the context of time-dependent route planning with additional time-invariant costs. Suffix-optimality of MC paths, however, is always provided.

Lemma 3.16. *If* $\langle u_1 \to_{f_1|c_1} \cdots \to_{f_{k-1}|c_{k-1}} u_k \rangle$ *is a* (u_1, u_k, τ_0)*-MC-path, then every suffix* $\langle u_i \to \cdots \to u_k \rangle$ *is a* $(u_i, u_k, \mathbf{arr}\, f_{\langle u_1 \to \cdots \to u_i \rangle}(\tau_0))$*-MC-path. In other words, we have* $\text{Cost}_G(u_1, u_k, \tau_0) = \text{Cost}_G(u_i, u_k, \mathbf{arr}\, f_{\langle u_1 \to \cdots \to u_i \rangle}(\tau_0)) + C_{\langle u_1 \to \cdots \to u_i \rangle}(\tau_0)$.

Proof. Assume the existence of a suffix that is no $(u_i, u_k, \mathbf{arr}\, f_{\langle u_1 \to \cdots \to u_i \rangle}(\tau_0))$-MC-path and replace it by a respective MC path (which exists due to Lemma 3.15) to obtain a contradiction. $\square$

In Section 3.1.3 we define upper and lower bounds for the minimum travel times. Now we do the same for minimum travel costs by defining

$$\begin{aligned} \overline{\text{Cost}}_G(s,t) &:= \min \left\{ \overline{C}_Q \,\middle|\, Q \text{ is path from } s \text{ to } t \text{ in } G \right\} \cup \{\infty\} , \\ \underline{\text{Cost}}_G(s,t) &:= \min \left\{ \underline{C}_Q \,\middle|\, Q \text{ is path from } s \text{ to } t \text{ in } G \right\} \cup \{\infty\} . \end{aligned} \tag{3.28}$$

Obviously, $\underline{\text{Cost}}_G(s,t) \le \text{Cost}_G(s,t,\tau) \le \overline{\text{Cost}}_G(s,t)$ holds for all $\tau \in \mathbb{R}$. Of course, $\text{Cost}_G(s,t,\cdot)$ forms a TCF; namely, the TCF that yields the minimum possible travel cost from s to t for all departure times. In fact, $\text{Cost}_G(s,t,\cdot)$ is the cost profile (CP) mentioned in Section 1.1.2. Obviously, $\text{Cost}_G(s,t,\tau_0)$ and $\text{Cost}_G(s,t,\cdot)$ are just what a minimum cost (MC) query and a CP query compute respectively.

The MC paths considered in this thesis can not change arbitrarily over time. More precisely, MC paths can only be valid for closed departure intervals or for isolated departure times. This generalizes an observation made by Foschini et al. in the context of EA paths (see Observation 2.3 in [36]), which we recapitulate in Section 5.2.1 (see Observation 5.2).

Lemma 3.17. *Consider an (s,t,τ_0)-MC-path P in G. Then, there is an interval $[a,b] \ni \tau_0$, such that P is either an (s,t,τ)-MC-path for all $\tau \in [a,b]$, or not an (s,t,τ)-MC-path for all $\tau \in [a,b] \setminus \{\tau_0\}$.*

Proof. $\mathrm{Cost}_G(s,t,\cdot)$ is the pointwise minimum of TCFs $C_{R_i} = f_{R_i} + c_{R_i}$ with $f_{R_i} \in \mathscr{F}_\Pi$ and $c_{R_i} \in \mathbb{R}_{\geq 0}$ for finitely many cycle-free paths $R_1,\ldots,R_k$ from s to t in G with $P = R_i$ for some i. All these TCFs are obviously continuous p.w.l. functions. Whether P is an MC path or not, can only change at departure times where at least two such TCFs C_{R_i} and C_{R_j} intersect. But as p.w.l. functions they can only intersect in isolated points or in line segments. $\square$

With a similar argument we obtain that $\mathrm{Cost}_G(s,t,\cdot)$ is the minimum of finitely many well-behaved TCFs and hence well-behaved itself.

Lemma 3.18. $\mathrm{Cost}_G(s,t,\cdot)$ *is a continuous p.w.l. function with period* Π. *Formally, it even fulfills the FIFO property; that is,* $\mathrm{Cost}_G(s,t,\cdot) \in \mathscr{F}_\Pi$.

Proof. We argue like in the proof of Lemma 3.17 that $\mathrm{Cost}_G(s,t,\cdot)$ is the minimum of the TCFs $C_{R_1},\ldots,C_{R_k}$ of finitely many paths $R_1,\ldots,R_k$, which are of the form $C_{R_i} = f_{R_i} + c_{R_i}$ with $C_{R_i} \in \mathscr{F}_\Pi$ for all $i \in \{1,\ldots,k\}$. But $\mathscr{F}_\Pi$ is closed under minimum (see Corollary 3.4), which implies $\mathrm{Cost}_G(s,t,\cdot) \in \mathscr{F}_\Pi$. $\square$

For every (s,t,τ_0)-MC-path P there is a corresponding travel time; namely, the travel time $f_P(\tau_0)$ of the path P. Accordingly, we define

$$\mathrm{MCTT}_G(s,t,\tau_0) := \lim_{\tau \to \tau_0^-} \min \big\{ f_Q(\tau) \,\big|\, Q \text{ is } (s,t,\tau)\text{-MC-path in } G \big\} \cup \{\infty\} \quad (3.29)$$

to denote the left-continuous minimum travel time of an (s,t,τ_0)-MC-path. Choosing the minimum is necessary in case of multiple (s,t,τ_0)-MC-paths. Note that $\mathrm{MCTT}_G(s,t,\cdot)$ is a TTF with

$$\mathrm{Cost}_G(s,t,\tau) \geq \mathrm{MCTT}_G(s,t,\tau) \geq \mathrm{TT}_G(s,t,\tau)$$

for all $\tau \in \mathbb{R}$. In general, $\mathrm{MCTT}_G(s,t,\cdot)$ is not continuous, but it is guaranteed to be left-continuous. The latter comes from the left-sided limit in Equation (3.29).[6]

[6]Whether points of discontinuity are treated as left- or right-continuous should depend on the application context. In this thesis, they are treated as left-continuous.

Note that $\mathrm{MCTT}_G(s,t,\cdot)$ yields the minimum possible travel time of a minimum cost path except for points of discontinuity. Otherwise left-continuity could not be guaranteed. However, every travel time $\mathrm{MCTT}_G(s,t,\tau)$ corresponds to a real (s,t,τ)-MC-path for all departure times $\tau \in \mathbb{R}$. This follows from the fact f_P is a continuous TTF for every path P in G.

Lemma 3.19. *For $s,t \in V$, where t is reachable from s in G,* $\mathrm{MCTT}_G(s,t,\cdot)$ *is a left-continuous real function with period* Π *that has only finitely many points of discontinuity on* $[0,\Pi)$.

Proof. That $\mathrm{MCTT}_G(s,t,\cdot)$ can have points of discontinuity becomes clear from the example in Figure 3.7. To show that $\mathrm{MCTT}_G(s,t,\cdot)$ is left-continuous with finitely many points of discontinuity, we argue as follows: From Lemma 3.6 and 3.13 we know that $\mathrm{MCTT}_G(s,t,\tau_0)$ arises from finitely many cycle-free paths in G. Together with Lemma 3.17, we obtain that $[0,\Pi)$ can be partitioned into finitely many subsets $\{0\},(0,a_1),\{a_1\},(a_1,a_2),\dots,\{a_{k-1}\},(a_{k-1},\Pi)$, such that paths $Q_1,\dots,Q_k$ exist that are (s,t,τ)-MC-paths with $\mathrm{MCTT}(s,t,\tau) = f_{Q_i}(\tau)$ for all $\tau \in (a_{i-1},a_i)$ with $1 \le i \le k$ (with $a_0 = 0$ and $a_k = \Pi$). But then, discontinuities can only occur at $0,a_1,\dots,a_{k-1}$ in $[0,\Pi)$. That $\mathrm{MCTT}_G(s,t,\cdot)$ is left-continuous, follows from its definition, because all TTFs f_{Q_i} are continuous. $\square$

Lemma 3.19 prompts us to define a generalized set of TTFs; namely the set $\mathscr{F}_\Pi^1$ of all functions $f : \mathbb{R} \to \mathbb{R}_{\ge 0}$ such that

- f has period Π and is left-continuous,
- $f|_{[0,\Pi)}$ has finitely many points of discontinuity $0 \le a_0 < \cdots < a_{k-1} < \Pi$,
- f is p.w.l. and continuous on $(a_0, a_1], (a_1,a_2], \dots, (a_{k-1}, a_0 + \Pi]$ each, and
- f fulfills the FIFO property on $(a_0, a_1], (a_1,a_2], \dots, (a_{k-1}, a_0 + \Pi]$ each.

So, all $f \in \mathscr{F}_\Pi^1$ are p.w.l. left-continuous TTFs with period Π and finitely many points of discontinuity on $[0,\Pi)$. The FIFO property may be violated at the points of discontinuity; namely, if $\lim_{\tau \to \tau_0^-} f(\tau) > \lim_{\tau \to \tau_0^+} f(\tau)$ holds.

3.2.2 Basic Operations with Additional Costs

Earlier in this thesis (see Section 3.1.2) we define three basic operations on TTFs; namely, evaluation, linking, and pointwise minimum. Here, we consider three analogous basic operations that we need if the edges of the road network have additional time-invariant costs. These operations have to deal both with travel times and travel costs. This is necessary, because time-dependent travel costs can only be interpreted properly together with the travel time. For this reason, we extend the basic operations not only to deal with TCFs but with a combination of TTFs and TCFs.

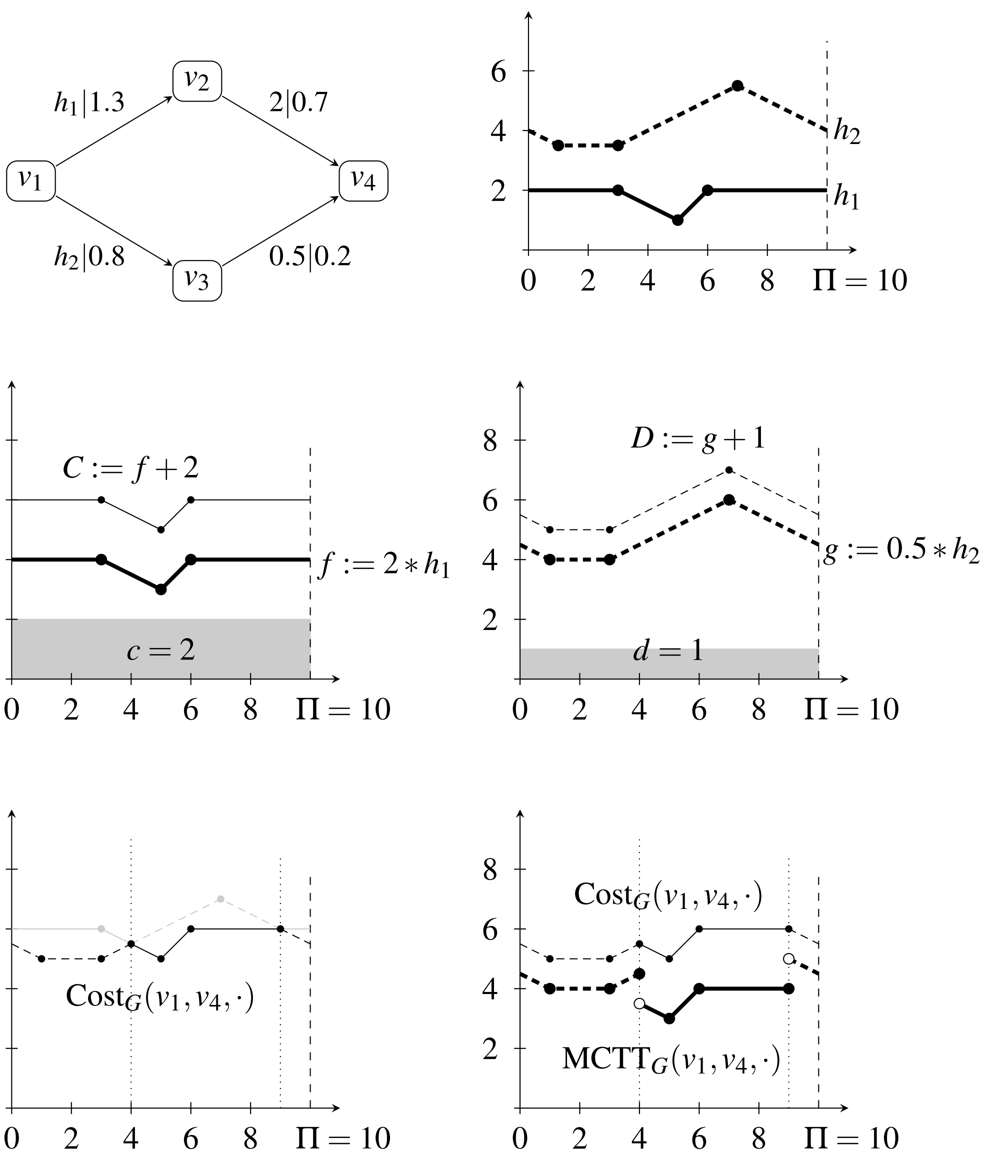


Figure 3.7. Example of a simple time-dependent road network G with additional time-invariant costs, where $\mathrm{Cost}_G(s,t,\cdot)$ has points of discontinuity. The TTFs $h_1, h_2 \in \mathscr{F}_{10}$ are non-constant (top). The TTFs of the paths $\langle v_1 \to v_2 \to v_4\rangle$ and $\langle v_1 \to v_3 \to v_4\rangle$ are the TTFs $f := 2 * h_1 = 2 + h_1$ and $g := 0.5 * h_2 = 0.5 + h_1$, respectively, the total additional costs of these paths amount to $c := 2 = 1.3 + 0.7$ and $d := 1 = 0.8 + 0.2$. The resulting TCFs of the two paths are $C := f + c$ and $D := g + d$ (middle). One sees that $\mathrm{Cost}_G(v_1, v_4, \cdot) = \min(C, D)$ is a continuous p.w.l. function with period 10 (bottom left). In contrast, $\mathrm{MCTT}_G(v_1, v_4, \cdot)$ has discontinuities at 4 and 9 (but it is still left-continuous). The reason is, that $\mathrm{MCTT}_G(v_1, v_4, \cdot)$ equals f on the interval $(4, 9]$ and g on the intervals $(0, 4]$ and $(9, 10]$ (bottom right).

For the computation of time-dependent MC paths as discussed in this thesis it is also not enough to consider TCFs of the form $f+c$ with TTF $f \in \mathscr{F}_\Pi$ and additional time-invariant cost $c \in \mathbb{R}_{\geq 0}$ only. Lemma 3.19 already suggests this. There, $\mathrm{Cost}_G(s,t,\cdot)$ is continuous, but $\mathrm{MCTT}_G(s,t,\cdot)$, which is a corresponding TTF in the sense that it yields the travel time on a corresponding MC path, is non-continuous. So, $\mathrm{Cost}_G(s,t,\cdot)$ cannot be the sum of a corresponding continuous TTF and a constant value. It turns out, however, that TCFs of the form $f+c$ are enough if the TTF f is allowed to have points of discontinuity (i.e., $f \in \mathscr{F}^1_\Pi$) and c is a non-negative *piecewise constant (p.w.c.)* real function. We call c an *additional cost function (ACF)*. We thus extend the basic operations to work on pairs $f|c$ with a TTF $f \in \mathscr{F}^1_\Pi$ and an ACF $c : \mathbb{R} \to \mathbb{R}_{\geq 0}$. The pair $f|c$ implicitly *represents* the TCF $f+c$. We write $\mathbf{tcf}\, f|c$ defined as $\mathbf{tcf}\, f|c := f+c$ to denote the represented TCF.

Piecewise Constant Functions. An ACF $c : \mathbb{R} \to \mathbb{R}_{\geq 0}$ is always taken from the set $\mathscr{X}_\Pi$ of *p.w.c. left-continuous real functions*. Every $c \in \mathscr{X}_\Pi$ fulfills the following conditions:

- $c : \mathbb{R} \to \mathbb{R}_{\geq 0}$ has period Π and is left-continuous,
- $c|_{[0,\Pi)}$ has finitely many points of discontinuity $0 \leq a_0 < \cdots < a_{k-1} < \Pi$, and
- c is constant on $(a_0, a_1], (a_1, a_2], \ldots, (a_{\ell-1}, a_0 + \Pi]$ each; that is, $c|_{(a_0, a_0+\Pi]}$ is of the form

$$c : \tau \mapsto \begin{cases} \gamma_0 & \text{if } \tau \in (a_0, a_1] \\ \gamma_1 & \text{if } \tau \in (a_1, a_2] \\ \vdots & \\ \gamma_{k-1} & \text{if } \tau \in (a_{k-1}, a_0 + \Pi] \end{cases} . \tag{3.30}$$

The use of p.w.c. functions is perfectly compatible with way how we denote time-dependent edges with additional time-invariant costs. We only have to identify the constant $c \in \mathbb{R}_{\geq 0}$ in $u \to_{f|c} v \in E$ with the constant function $\tau \mapsto c$. We write $|c|$ to denote the *complexity* of c. That is, $|c| := k$ is the number discontinuities that c has on the interval $[0, \Pi)$. Note that we set $|c| = 1$ if c is constant, although c is actually continuous in this case.

Evaluation with Piecewise Constant Additional Costs. Given $f|c \in \mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$ as well as a departure time τ_0 we want to compute $\mathbf{tcf}(f|c)(\tau_0) = f(\tau_0) + c(\tau_0)$. Assume we represent $f|c$ as sequence

$$\langle (f^{(0)}, \gamma_0), (f^{(1)}, \gamma_1), \ldots, (f^{(k-1)}, \gamma_{k-1}) \rangle \tag{3.31}$$

with continuous partial real functions $f^{(0)}, \ldots, f^{(k-1)}$ and $\gamma_0, \ldots, \gamma_{k-1} \in \mathbb{R}_{\geq 0}$, such that $\mathrm{dom}\, f_i = (a_i, a_{i+1}]$ holds with $a_k = a_0 + \Pi$ and $0 \leq a_0 < \cdots < a_{k-1} < \Pi$. Then,

$f(\tau_0)$ and $c(\tau_0)$ can be obtained by choosing $(f^{(i)}, \gamma_i)$ with $(\tau_0 \bmod \Pi) \in \operatorname{dom} f^{(i)}$ utilizing that $f(\tau_0) = f(\tau_0 \bmod \Pi)$ and $c(\tau_0) = c(\tau_0 \bmod \Pi)$ hold. So, $c(\tau_0) = \gamma_i$ can be obtained directly and $f(\tau_0)$ can be computed by linear interpolation (see Section 3.1.2).

Linking with Piecewise Constant Additional Costs. Given two adjacent edges $u \rightarrow_{f|c} v$ and $v \rightarrow_{g|d} w$, with $f|c, g|d \in \mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$, we want to know the TTF as well as the ACF of the whole path $\langle u \rightarrow_{f|c} v \rightarrow_{g|d} w\rangle$.

We first consider the TCF. Traveling along the edge $u \rightarrow_{f|c} v$ simply costs $f(\tau) + c(\tau)$ for τ being the time we depart from u. Traveling along $v \rightarrow_{g|d} w$ then costs $(g+d)(\mathbf{arr}\, f(\tau)) = g(\mathbf{arr}\, f(\tau)) + d(\mathbf{arr}\, f(\tau))$, because $\mathbf{arr}\, f(\tau)$ is the time we arrive at v and also the time we depart from this node (remember that waiting does not result in a lower travel cost, see Lemma 3.14). So, traveling the whole path costs

$$\begin{aligned}(g+d)(\mathbf{arr}\, f(\tau)) + (f+c)(\tau) &= g(\mathbf{arr}\, f(\tau)) + d(\mathbf{arr}\, f(\tau)) + f(\tau) + c(\tau)\\ &= g(\mathbf{arr}\, f(\tau)) + f(\tau) + d(\mathbf{arr}\, f(\tau)) + c(\tau)\\ &= g * f(\tau) + d(\mathbf{arr}\, f(\tau)) + c(\tau)\ .\end{aligned}$$

Obviously, $g * f$ is the TTF of the whole path. We correspondingly define

$$d *_f c := d \circ \mathbf{arr}\, f + c \tag{3.32}$$

to denote the p.w.c. ACF along the whole path. This justifies the definition

$$g|d \star f|c := g * f \,|\, d *_f c \tag{3.33}$$

to obtain an extension of the linking operation that works with $\mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$. The TCF represented by $g * f \,|\, d *_f c$ is

$$g * f + d *_f c = (g+d) \circ \mathbf{arr}\, f + f + c\ .$$

Lemma 3.20. *The set $\mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$ is closed under linking; that is, $f|c, g|d \in \mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$ implies $g|d \star f|c \in \mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$.*

Proof. Lemma 2.1 tells us that compositions of left-continuous functions are again left-continuous. Further, let $\{\tau_0, \dots, \tau_{k-1}\}$, $\{\gamma_0, \dots, \gamma_{\ell-1}\}$ and $\{\delta_0, \dots, \delta_{m-1}\}$ be all the points of discontinuity, respectively, that f, c and d have on $[0, \Pi)$. Then, $d *_f c = d \circ \mathbf{arr}\, f + c$ can have no other points of discontinuity than $\{\tau_0, \dots, \tau_{k-1}, \gamma_0, \dots, \gamma_{\ell-1}\}$ as well as $\{\max \mathbf{dep}\, f(\delta_0), \dots, \max \mathbf{dep}\, f(\delta_{m-1})\}$, which are finitely many. So, $d *_f c \in \mathscr{X}_\Pi$ follows easily, $g * f \in \mathscr{F}^1_\Pi$ holds analogously. □

Linking with additional cost fulfills the law of associativity. More precisely, $f|c,\, g|d,\, h|e \in \mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$ always fulfill

$$h|e \star (g|d \star f|c) = (h|e \star g|d) \star f|c\ . \tag{3.34}$$

To prove that we calculate

$$\begin{aligned} h|e \star (g|d \star f|c) &= h|e \star \big(g*f \,\big|\, d *_f c\big) = h*g*f \,\big|\, e *_{g*f} (d *_f c) \\ &= h*g*f \,\big|\, e \circ \mathbf{arr}(g*f) + d \circ \mathbf{arr}\, f + c \\ &= h*g*f \,\big|\, e \circ \mathbf{arr}\, g \circ \mathbf{arr}\, f + d \circ \mathbf{arr}\, f + c \end{aligned}$$

as well as

$$\begin{aligned} (h|e \star g|d) \star f|c &= \big(h*g \,\big|\, e *_g d\big) \star f|c = h*g*f \,\big|\, (e *_g d) *_f c \\ &= h*g*f \,\big|\, (e \circ \mathbf{arr}\, g + d) \circ \mathbf{arr}\, f + c \\ &= h*g*f \,\big|\, e \circ \mathbf{arr}\, g \circ \mathbf{arr}\, f + d \circ \mathbf{arr}\, f + c\ . \end{aligned}$$

It is interesting to note that the “second component” of the generalized linking (i.e., the linking of the p.w.c. additional costs) is *not* associative. More precisely, only a variety of associativity is fulfilled there; namely,

$$e *_{g*f} (d *_f c) = (e *_g d) *_f c\ . \tag{3.35}$$

Generalizing the Notation. So far, the symbols f_P, c_P, C_P, $\underline{C}_P$, $\overline{C}_P$, with P being a path, as well as $\mathrm{Cost}_G(s,t,\cdot)$, $\mathrm{MCTT}_G(s,t,\cdot)$, $\underline{\mathrm{Cost}}_G(s,t)$, and $\overline{\mathrm{Cost}}_G(s,t)$, with s and t being nodes, are only defined in the context edge weights taken from $\mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$ (see Section 3.1.3 and 3.2.1). Having generalized the linking operation, we are now able to define these symbols to work with edge weights taken from $\mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$ instead of $\mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$. There, it must be noted that such generalized edge weights and the corresponding shortest path structure are not a main issue of this thesis. But we have to discuss them because one of the route planning techniques described in this thesis needs to deal with this kind of edge weights to work properly (see Chapter 6).

Consider a path $P = \langle u_1 \to_{f_1|c_1} \cdots \to_{f_{k-1}|c_{k-1}} u_k \rangle$ in G with $f_i \in \mathscr{F}^1_\Pi$ and $c_i \in \mathscr{X}_\Pi$ for all $i = 1,\dots,k-1$. The original definition of f_P (see Equation (3.22)), which defines f_P in terms of TTFs taken from $\mathscr{F}_\Pi$, is perfectly compatible with TTFs taken from $\mathscr{F}^1_\Pi$. In case of the additional cost c_P things are a little more complicated because the ACFs $c_1,\dots,c_{k-1}$ cannot be linked conveniently without considering the TTFs $f_1,\dots,f_{k-2}$. For example, we could write

$$c_P = c_{k-1} *_{f_{k-2}*\cdots*f_1} \big(c_{k-2} *_{f_{k-3}*\cdots*f_1} \big(\dots c_3 *_{f_2*f_1} (c_2 *_{f_1} c_1) \dots\big)\big)\ ,$$

which is very complicated and the brackets cannot be omitted. Due to Equation (3.35) we could also write

$$c_P = \big(\big(\dots(c_{k-1} *_{f_{k-2}} c_{k-2}) *_{f_{k-3}} \dots c_3\big) *_{f_2} c_2\big) *_{f_1} c_1 \ ,$$

which is a little less complicated, and the brackets cannot be omitted, too.

So, we utilize the generalized linking operation defined before to obtain a generalized definition of f_P and c_P by setting

$$f_P|c_P := f_{k-1}|c_{k-1} \star \dots \star f_1|c_1 \ . \tag{3.36}$$

Still, $C_P = f_P + c_P$ yields the TCF of the path P, but note that the original definition of C_P (see Equation (3.26) in Section 3.2.1) only covers additional time-invariant costs (i.e., $c_P \in \mathbb{R}_{\geq 0}$). We generalize this by defining

$$C_P(\tau) := f_P(\tau) + c_P(\tau) \ , \tag{3.37}$$

which is perfectly compatible with Equation (3.26). Generalizing $\underline{C}_P$ and $\overline{C}_P$, the only problem is that piecewise continuous TTFs do not have a well-defined minimum or maximum in general. To fix that, we use infimum and supremum, which are always well-defined, and redefine

$$\begin{aligned} \underline{C}_P &:= \inf f_1 + \dots + \inf f_{k-1} + \min c_1 + \dots + \min c_{k-1} \ , \\ \overline{C}_P &:= \sup f_1 + \dots + \sup f_{k-1} + \max c_1 + \dots + \max c_{k-1} \ . \end{aligned} \tag{3.38}$$

Then, the definitions of $\underline{\mathrm{Cost}}_G(s,t)$ and $\overline{\mathrm{Cost}}_G(s,t)$ (see Equation (3.28)) generalize in a natural way. Regarding the redefinition of $\mathrm{Cost}_G(s,t,\cdot)$ and $\mathrm{MCTT}_G(s,t,\cdot)$, the question arises whether an (s,t,τ)-MC path actually exists for all $s,t \in V$ and $\tau \in \mathbb{R}$. In fact, we strongly conjecture that this is the case but have not worked out a proof so far. A simple condition that should always be fulfilled in practice, however, guarantees the existence of MC paths easily (see Lemma 3.23 in Section 3.2.4).

Minimum with Piecewise Constant Additional Costs. Consider two paths P and P' that both go from u to v. Let $f := f_P$, $g := f_{P'} \in \mathscr{F}^1_\Pi$ be the TTFs and $c := c_P$, $d := c_{P'} \in \mathscr{X}_\Pi$ be ACFs of these paths respectively. We want to know a "common TTF" and a "common ACF" that together represent the "common TCF" of these two paths. The latter means the TCF describing how expensive it is to travel from u to v, if we are allowed to travel along P or P' always choosing the better path for every departure time. The resulting merged TCF is just the pointwise minimum of $f+c$ and $g+d$; namely,

$$\min(f+c, g+d) \ .$$

To obtain a "common TTF" we define

$$\min_{c,d}(f,g) : \tau \mapsto \lim_{\sigma \to \tau^-} \begin{cases} f(\sigma) & \text{if } f + c(\sigma) < g + d(\sigma) \\ g(\sigma) & \text{if } f + c(\sigma) > g + d(\sigma) \\ \min(f,g)(\sigma) & \text{if } f + c(\sigma) = g + d(\sigma) \end{cases} , \tag{3.39}$$

which yields the minimum possible left-continuous travel time of traveling along P or P' at minimum cost for every departure time τ. At points of discontinuity, $\min_{c,d}(f,g)$ is not guaranteed to yield the minimum travel time of such a minimum cost travel. However, it is always $\min_{c,d}(f,g)(\tau_0) = f_P(\tau_0)$ or $\min_{c,d}(f,g)(\tau_0) = f_{P'}(\tau_0)$, even if $\min_{c,d}(f,g)$ is discontinuous at τ_0.

A "common ACF" of a minimum cost travel along P or P', that corresponds to the travel time given by $\min_{c,d}(f,g)$, amounts to

$$\min^{f,g}(c,d) := \min(f+c,\, g+d) - \min_{c,d}(f,g) \; . \tag{3.40}$$

Note that $\min^{f,g}(c,d)$ is left-continuous too, just like $\min_{c,d}(f,g)$. There is a more explicit construction of $\min^{f,g}(c,d)$ than the definition in Equation (3.40).

Lemma 3.21. *For $f,g \in \mathscr{F}^1_\Pi$ and $c,d \in \mathscr{X}_\Pi$ we have*

$$\min^{f,g}(c,d)(\tau) = \lim_{\sigma \to \tau^-} \begin{cases} c(\sigma) & \text{if } f + c(\sigma) < g + d(\sigma) \\ d(\sigma) & \text{if } f + c(\sigma) > g + d(\sigma) \\ \max(c,d)(\sigma) & \text{if } f + c(\sigma) = g + d(\sigma) \end{cases} . \tag{3.41}$$

Proof. Utilizing that f, g, c and d are all left-continuous, we consider three different cases. First, we assume $f + c(\sigma) < g + d(\sigma)$ holds for all $\sigma \in (\tau - \delta, \tau)$ and some $\delta > 0$. Then, we calculate

$$\begin{aligned} \min^{f,g}(c,d)(\tau) &= \min(f+c,\, g+d)(\tau) - \lim_{\sigma \to \tau^-} f(\sigma) \\ &= \lim_{\sigma \to \tau^-} f(\sigma) + c(\sigma) - f(\sigma) \\ &= \lim_{\sigma \to \tau^-} c(\sigma) \; , \end{aligned}$$

because f and c are left-continuous. Second, we assume $f + c(\sigma) > g + d(\sigma)$ for all $\sigma \in (\tau - \delta, \tau)$ and obtain $\min^{f,g}(c,d)(\tau) = \lim_{sigma \to \tau^-} d(\tau)$ analogously. Third, we assume $f + c(\sigma) = g + d(\sigma)$ for all $\sigma \in (\tau - \delta, \tau)$. Also, we assume w.l.o.g. that $c(\sigma) \le d(\sigma)$ holds. But this implies $f(\sigma) \ge g(\sigma)$ and we calculate

$$\begin{aligned} \min^{f,g}(c,d)(\tau) &= \min(f+c,\, g+d)(\tau) - \lim_{\sigma \to \tau^-} \min\{f(\sigma), g(\sigma)\} \\ &= \lim_{\sigma \to \tau^-} g(\sigma) + d(\sigma) - g(\sigma) \\ &= \lim_{\sigma \to \tau^-} \max\{c(\sigma), d(\sigma)\} \end{aligned}$$

and are finished. □

We now extend the minimum operation to work with $\mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$ and define

$$\min(f|c,\, g|d) := \min_{c,d}(f,g) \,|\, \min^{f,g}(c,d)\ . \tag{3.42}$$

This means

$$\textbf{tcf}\min(f|c,\, g|d) = \min_{c,d}(f,g) + \min^{f,g}(c,d) = \min(f+c,\, g+d)$$

is the TCF for traveling along P or P' while always choosing the cheaper path for every departure time.

Lemma 3.22. *The set $\mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$ is closed under minimum; that is, $f|c,\, g|d \in \mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$ implies $\min(g|d,\, f|c) \in \mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$.*

Proof. The left-continuity of $\min_{c,d}(f,g)$ follows directly from the definition in Equation (3.39), because f,g,c,d are already left-continuous. Then, $\min_{c,d}(f,g) \in \mathscr{F}^1_\Pi$ follows easily. With Lemma 3.21 we obtain $\min^{f,g}(c,d) \in \mathscr{X}_\Pi$ analogously. $\square$

The Minimum of TTFs with additional p.w.c. costs behaves well in the sense that the associative and the commutative property is are fulfilled; that is,

$$\min(\min(f|c,\, g|d),\, h|e) = \min(f|c,\, \min(g|d,\, h|e))\ , \tag{3.43}$$

$$\min(f|c,\, g|d) = \min(g|d,\, f|c)\ . \tag{3.44}$$

Both can be verified easily. However, the linking operation "$\star$" does in general not distribute over the minimum, which is unlike the minimum on TTFs alone. At least, right-distributivity is fulfilled, a property inherited from Equation (2.7). More precisely,

$$\min(f|c,\, g|d) \star h|e = \min(f|c \star h|e,\ g|d \star h|e) \tag{3.45}$$

holds for all $f|c,\, g|d,\, h|e \in \mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$. To understand that, we calculate

$$\begin{aligned}
&\min_{c *_h e,\, d *_h e}(f*h,\, g*h) \\
&\qquad = \min_{c \circ \mathbf{arr}\, h + e,\, d \circ \mathbf{arr}\, h + e}(f \circ \mathbf{arr}\, h + h,\, g \circ \mathbf{arr}\, h + h) \\
&\qquad = \min_{c \circ \mathbf{arr}\, h,\, d \circ \mathbf{arr}\, h}(f \circ \mathbf{arr}\, h,\, g \circ \mathbf{arr}\, h) + h \\
&\qquad = \min_{c,d}(f,g) \circ \mathbf{arr}\, h + h \\
&\qquad = \min_{c,d}(f,g) * h\ .
\end{aligned}$$

with respect to Equation (3.39). Analogously, Equation (3.41) yields

$$\min^{f*h,\, g*h}(c *_h e,\, d *_h e) = \min^{f,g}(c,d) *_h e\ .$$

3.2.3 Computing with Additional Costs Efficiently

Section 3.2.2 extends the three basic operations (evaluation, linking, and minimum) to work with pairs of TTFs and ACFs $f|c \in \mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$. Here, we discuss how these three operations can be computed efficiently.

Efficient Evaluation with Additional Costs. Evaluating a pair $f|c \in \mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$ for a given $\tau_0 \in \mathbb{R}$ (see page 104) can be done by applying binary search to the sequence

$$\langle (f^{(0)}, \gamma_0), (f^{(1)}, \gamma_1), \ldots, (f^{(k-1)}, \gamma_{k-1}) \rangle$$

that represents $f|c$ just like in Equation (3.31). But in our experience, c has usually few points of discontinuity. So, one can store the ACF c separately as a (supposedly short) sequence

$$\langle (\xi_0, \gamma_0), (\xi_1, \gamma_1), \ldots, (\xi_{\ell-1}, \gamma_{\ell-1}) \rangle \ , \tag{3.46}$$

such that $c(\tau) = \gamma_i$ for all $\tau \in (\xi_i, \xi_{i+1}]$. To compute $c(\tau_0)$ one can choose the appropriate interval $(\xi_i, \xi_{i+1}]$ by a linear scan[7] and return γ_i as result. To compute $f(\tau_0)$, one can store the TTF f as sequence of bend points

$$\langle (x_0, y_0), (x_1, y_1), \ldots, (x_{m-1}, y_{m-1}) \rangle \tag{3.47}$$

as in Equation (3.1) and use the bucket data structure described in the context of evaluating TTFs (see page 91) for fast evaluation. The only problem is how to handle points of discontinuity. But this can be done easily by allowing $x_i = x_{i+1}$ if f is discontinuous at x_i (in the context of continuous TTFs we require $x_i < x_{i+1}$).

Efficient Minimum with Additional Costs. A method to compute the generalized minimum $\min(f|c, g|d)$ for given $f|c, g|d \in \mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$ can be obtained by generalizing how the minimum of TTFs is computed (see Algorithm 3.2 in Section 3.1.4). To do so, one represents $f|c$ and $g|d$ as described by Equations (3.46) and (3.47). More precisely, $f|c$ is represented as a pair of sequences

$$\Big(\big\langle (\xi_0, \gamma_0), \ldots, (\xi_{\ell-1}, \gamma_{\ell-1}) \big\rangle, \big\langle (x_0, y_0), \ldots, (x_{m-1}, y_{m-1}) \big\rangle \Big) \ ,$$

where the first sequence represents the TTF f and the second represents the ACF c. If $g|d$ is represented analogously as

$$\Big(\big\langle (\zeta_0, \delta_0), \ldots, (\zeta_{k-1}, \delta_{k-1}) \big\rangle, \big\langle (x'_0, y'_0), \ldots, (x'_{n-1}, y'_{n-1}) \big\rangle \Big) \ ,$$

then $\min(f|c, g|d)$ can be computed by simultaneously scanning these four sequences from left to right, which takes $\mathrm{O}(|f| + |c| + |g| + |d|)$ time. The details are left to the reader.

[7] Though a linear scan takes linear time in theory, it is very cache efficient on arrays. So, it can be performed very fast in practice if the sequence is not too long.

Efficient Linking with Additional Costs. The generalized linking $g|d \star f|c$ with $f|c, g|d \in \mathscr{F}_\Pi^1 \times \mathscr{X}_\Pi$ can be computed by generalizing the algorithm for linking TTFs described in Section 3.1.4 (see Algorithm 3.1). This is analogous to the minimum operation, where we also generalize the algorithm for computing the minimum of TTFs. In case of linking, however, things are more complicated and the resulting algorithm has superlinear running time.

Again, $f|c$ and $g|d$ are represented as pairs of sequences as described by Equations (3.46) and (3.47). That is, $f|c$ is represented as

$$\Big(\big\langle(\xi_0,\gamma_0),\dots,(\xi_{\ell-1},\gamma_{\ell-1})\big\rangle, \big\langle(x_0,y_0),\dots,(x_{m-1},y_{m-1})\big\rangle\Big)\ ,$$

and $g|d$ is represented as

$$\Big(\big\langle(\zeta_0,\delta_0),\dots,(\zeta_{k-1},\delta_{k-1})\big\rangle, \big\langle(x'_0,y'_0),\dots,(x'_{n-1},y'_{n-1})\big\rangle\Big)\ .$$

The first sequence is the TTF and the second sequence is the ACF each. Analogously to Algorithm 3.1, $g|d \star f|c$ can be computed by scanning the four sequences. However, if f has negative discontinuities (see Section 2.1.1), we may have to jump back when scanning through g and d. This is because of $g|d \star f|c = g * f|d *_f c = g \circ \mathbf{arr} f + f|d \circ \mathbf{arr} f + c$, which means that the scanning through g and d is governed by the scanning through $\mathbf{arr} f$. So, if f has a negative discontinuity, the same holds true for $\mathbf{arr} f$, and we have to go back.

In the worst case, f has $\Theta(|f|)$ negative discontinuities on $[0,\Pi)$. Moreover, in the extreme case, f may increase arbitrarily heavily between two points of discontinuity, which means that g and d must be scanned L/Π times, for $L = f(b) - \lim_{\tau\to a^+} f(\tau)$ being the increase of f between two neighboring points of discontinuity a and b with $a < b$. This leads to a worst-case running time of

$$\Theta\Big(\big(|g| + |d|\big) \cdot \frac{L_f \cdot |f|}{\Pi} + |c|\Big)$$

with $L_f := \max f - \min f$, because f and c are only scanned once, g and d may be scanned L_f times for any negative discontinuity of f.

In the context of road networks with additional time-invariant costs, however, things are better because $C_P = f_P + c_P \in \mathscr{F}_\Pi$ is guaranteed for every path P, even if f_P and c_P alone have discontinuities. That is, $f_P + c_P$ is continuous and formally fulfills the FIFO property. As a consequence, a TTF f can never increase more than Π between two discontinuities in this setup. This means that g and d are scanned at most once for every bend point of f. So, the worst-case running time can be bounded by

$$\mathrm{O}\Big(\big(|g| + |d|\big) \cdot |f| + |c|\Big)\ .$$

As in case of the generalized minimum operation, we omit the details of the generalized linking algorithm and leave them to the reader.

3.2.4 MC Paths with Piecewise Constant Additional Costs

The generalized basic operations not only work with constant (i.e., time-invariant) but also with piecewise constant (i.e., time-dependent) additional costs. This raises the question what properties MC paths have if edge weights are taken from $\mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$ instead of $\mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$.

It is relatively obvious that waiting can be beneficial in this setup. To understand that, take a look at the TCF in Figure 3.8. This TCF has a negative discontinuity, which violates the FIFO property and makes waiting beneficial. If waiting is forbidden, then cycles can still be beneficial. Assume a cycle $\langle u \to \cdots \to u\rangle$ is present at the node u with $\tau' = \mathbf{arr}\, f_{\langle u\to\cdots\to u\rangle}(\tau)$. If $c_{\langle u\to\cdots\to u\rangle}(\tau)$ is sufficiently small, then we have

$$\begin{aligned} &C\big(\mathbf{arr}\, f_{\langle u\to\cdots\to u\rangle}(\tau)\big) + f_{\langle u\to\cdots\to u\rangle}(\tau) + c_{\langle u\to\cdots\to u\rangle}(\tau) \\ &\qquad\qquad = C(\tau') + \tau' - \tau + c_{\langle u\to\cdots\to u\rangle}(\tau) < C(\tau) \end{aligned}$$

and by traveling the cycle we save some cost.

Note that time-dependent road networks with edge weights taken from $\mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$ (i.e., with additional time-dependent costs) are not a primary issue of this thesis. Instead, we only deal with MC queries for edge weights from $\mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$ (i.e., with additional time-invariant costs), which is already difficult enough. The route planning technique for fast MC queries discussed in Chapter 6, however, needs to deal with edge weights taken from $\mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$ and forbidden waiting; regardless whether additional costs of the underlying road network are time-dependent or not. Although we strongly conjecture that (for given $s,t \in V$, $\tau_0 \in \mathbb{R}$) there always exists an MC paths in this setup, we have not worked out a proof so far. But there is a simple condition that guaranties the existence of MC paths easily.

Lemma 3.23. *Consider a graph G with edge weights taken from $\mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$ and $s,t \in V$, $\tau_0 \in \mathbb{R}$ where t is reachable from s. If $\inf f > 0$ or $\min c > 0$ holds for all $u \to_{f|c} v \in E$, then G contains an (s,t,τ_0)-MC-path—though maybe with cycles.*

Proof. Consider an arbitrary path $P = \langle s \to \cdots \to t\rangle \subseteq G$. Set

$$C_{\min} := \min\big\{ \inf f + \min c \;\big|\; u \to_{f|c} v \in E \big\} ,$$

which fulfills $C_{\min} > 0$. A path $R = \langle s \to \cdots \to t\rangle$ with $|R| > \sup C_P / C_{\min}$ fulfills $\inf C_R \geq |R| \cdot C_{\min} > \sup C_P$. So, amongst all paths from s to t, only the ones with $\lfloor \sup C_P / C_{\min} \rfloor$ or less hops (i.e., finitely many) can have a travel cost of $\sup C_P$ or less for any departure time. That means, an (s,t,τ_0)-MC-path exists in G. $\square$

The condition $\inf f > 0$ or $\min c > 0$ for all $u \to_{f|c} v$ should always be fulfilled in practice. Note that we write $\inf f$ and $\sup f$ for TTFs $f \in \mathscr{F}^1_\Pi$ instead of $\min f$

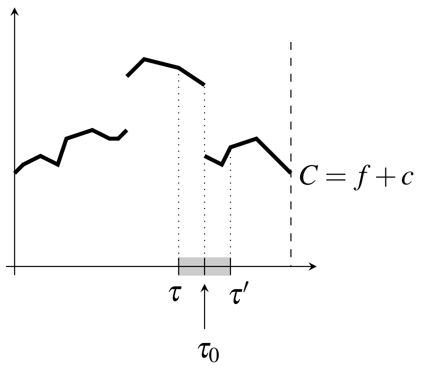


Figure 3.8. How waiting gets beneficial if edge weights are taken from $\mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$ instead of $\mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$. Assume the TCF $C = f + c$ of an edge $u \rightarrow_{f|c} v$ with $f|c \in \mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$ has a negative discontinuity at τ_0. Arriving in the node u at time $\tau \leq \tau_0$, it pays off to wait until $\tau' > \tau_0$ before traveling along $u \rightarrow_{f|c} v$. This is because of $C(\tau') + \tau' - \tau < C(\tau)$.

and $\max f$ respectively, because f has points of discontinuity in general and may not have a defined minimum or maximum hence. The statement of Lemma 3.23 is somewhat similar to a "finiteness criterion" by Orda and Rom (see Lemma 2 in their article [70]). There, the TCF C_e of each edge e also must fulfill $\min C_e > 0$. Note that Orda and Rom also consider waiting at nodes, which we do not allow.

Provided that the condition from Lemma 3.23 is fulfilled, we are able to prove a generalized version of Lemma 3.17.

Lemma 3.24. *Consider a graph G with $f|c \in \mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$ and $\inf(f + c) > 0$ for all $u \rightarrow_{f|c} v \in E$. For every (s,t,τ_0)-MC-path $P \subseteq G$, there is $\delta > 0$ such that P is an MC path either for all $\tau \in (\tau_0 - \delta, \tau_0)$ or for <u>no</u> $\tau \in (\tau_0 - \delta, \tau_0)$.*

Proof. Considering the proof of Lemma 3.23, we find that $\mathrm{Cost}_G(s,t,\cdot)$ is the pointwise minimum of finitely many left-continuous TCFs, and C_P is one of them. So, whether P is an MC path or not can only change at departure times where two such TCFs touch (including intersections) or at a point of discontinuity. That two TCFs touch is only possible for isolated departure times and on intervals. Moreover, there are only finitely many points of discontinuity. □

This enables us to show the guaranteed existence of MC paths that are not only valid on isolated departure times but on a right-closed and left-open interval.

Lemma 3.25. *Consider a graph G with $f|c \in \mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$ and $\inf(f + c) > 0$ for all $u \rightarrow_{f|c} v \in E$. For $s,t \in V, \tau_0 \in \mathbb{R}$ there is always a path $P = \langle s \rightarrow \cdots \rightarrow t \rangle$ in G, such that $C_P(\tau) = \mathrm{Cost}_G(s,t,\tau)$ for all $\tau \in (\tau_0 - \delta, \tau_0]$ with some $\delta > 0$.*

Proof. Lemma 3.23 ensures that one or more (s,t,τ_0)-MC-paths exist in G and considering the proof of this lemma we also find that $\mathrm{Cost}_G(s,t,\cdot)$ is the pointwise minimum of finitely many left-continuous TCFs $C_{P_1}, \dots, C_{P_k}$ of paths $P_1, \dots, P_k$ in G. So, $\delta > 0$ exists with $C_{P_i}(\tau) = \mathrm{Cost}_G(s,t,\tau)$ for all $\tau \in (\tau_0 - \delta, \tau_0)$ with some $i \in \{1, \dots, k\}$. However, C_{P_i} and $\mathrm{Cost}_G(s,t,\cdot)$ are left-continuous, because $\mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$ is closed under linking and minimum (see Lemma 3.20 and 3.22). So, $C_{P_i}(\tau_0) = \lim_{\tau \to \tau_0^-} C_{P_i}(\tau) = \lim_{\tau \to \tau_0^-} \mathrm{Cost}_G(s,t,\tau) = \mathrm{Cost}_G(s,t,\tau_0)$ holds. $\square$

3.3 References

This chapter is partly based on a journal article published together with Geisberger, Sanders and Vetter [8] and also on two conference articles published together with Geisberger, Neubauer, and Sanders [6] as well as with Sanders [9]. Several wordings of these articles have been used or rephrased. Especially, this applies to basic definitions, like earliest arrival (EA) and minimum cost (MC) paths for example.

The detailed discussion of the properties of MC paths with additional time-invariant (see Section 3.2.1) and time-dependent (see Section 3.2.4) costs has been worked out newly for this thesis. This is also the case for the algebraic structure $(\mathscr{F}^1_\Pi \times \mathscr{X}_\Pi, \star, \min)$ (see Section 3.2.2). The efficient computation of the basic operations on TTFs (see Section 3.1.4) is also new but relatively straightforward. It may, nevertheless, be interesting for people who want to put time-dependent route planning algorithms into practice.

Much of the material in Section 3.1 is not new. The fundamental properties of time-dependent minimum travel time paths (see Section 3.1.3) are already discussed by Dean [17]. Orda and Rom [69] already consider functions as first-class citizens, but their notation is still in terms of function evaluation with real-valued arguments. Arrival time functions and departure time functions of edges and paths (as in Section 3.1.1) have also been used by Dean in the past [17]. Explicit algebraical operations on TTFs (as in Section 3.1.2) have become common in the last few years (see Delling [19] or Batz et al. [6] for example).[8] Using the equivalent arrival time functions with function composition as linking operator is also common (see Foschini et al. [36]).

[8]Delling usually writes $f \oplus g$ instead of $g * f$ to denote linking of TTFs f, g.

4

Algorithmic Ingredients

This chapter describes the basic time-dependent shortest path algorithms utilized by TCHs and provides proofs of correctness for all of them. These algorithms are all variants of Dijkstra's algorithm and have similar structure hence. This means they all grow a search space from the start node while repeatedly processing and updating node labels and predecessor information by relaxing edges. We begin this chapter with a closer look at the common structure of these *Dijkstra-like* algorithms (see Section 4.1).

Most of the basic Dijkstra-like algorithms in this thesis only assign a single label to every node. Some of these basic *single-label searches* run in *forward* direction and some in *backward* direction. We provide detailed descriptions and proofs both for forward (see Section 4.2) and backward searches (see Section 4.3). Two Dijkstra-like algorithms, however, assign multiple labels to a node. This is similar to the basic multi-label search described earlier in Section 2.2.6. We provide descriptions and proofs of correctness for these time-dependent *multi-label searches*, too (see Section 4.4).

4.1 Common Structure of Dijkstra-like Algorithms

The basic Dijkstra-like algorithms considered in this thesis can be distinguished from one another by a few behavioral characteristics. These characteristics are described in Section 4.1.1. Dijkstra-like algorithms compute information not only about time-dependent travel times and costs but also about the corresponding optimal routes. This information is stored by the predecessor information set up during the computation. It implicitly represents a *predecessor graph*. This is explained in Section 4.1.2.

4.1.1 Characterizing Different Dijkstra-like Algorithms

Dijkstra-like search algorithms all have a very similar structure: Just like the original Dijkstra search (see Section 2.2.3), they grow a search space from the start node s while repeatedly processing and updating node labels and predecessor information by relaxing edges. Usually, a label of a node u represents some kind of minimum distance from s to u. The predecessor information of u stores one or more predecessor nodes of u on the corresponding optimal paths. In case of single-label search, nodes can have two possible states during computation: They are either *active* or *inactive*. The outgoing edges $u \to v$ of a node u are relaxed whenever the current label of the node u is somehow minimal amongst all labels of currently active nodes. After relaxing all its edges the node u gets inactive. In case of multi-label search it is similar, but with the difference that it is not the nodes but the labels that are either active or inactive.

The final node labels and the final predecessor information represent the desired result of a Dijkstra-like shortest path computation. The final predecessor information of all nodes implicitly represents a subgraph that contains one or more optimal paths from s to every reachable node (see Section 4.1.2). It is, in fact, a generalization of the predecessor tree mentioned in Section 2.2.3

A Dijkstra-like computation is usually controlled by a priority queue (PQ, see Section 2.2.2). More precisely, the distinction of active and inactive nodes (or labels) is usually determined by the membership in the PQ. A node (or label) that has minimal key with respect to the PQ is the node (or label) whose edges are relaxed next. The common structure of Dijkstra-like algorithms suggests the following few characteristics by that the different algorithms can be distinguished:

1. What is a node label?
2. Has a node a single label or multiple labels?
3. What is the structure of the predecessor information?
4. What is the key of a node or label with respect to the PQ?
5. How are the node labels updated on edge relaxation?
6. How is the predecessor information updated on edge relaxation?
7. How is the start node labeled initially?
8. Can the state of a node or label change from inactive to active multiple times?

The 8th item in the above list is, at least in case of single-label searches, as good as to ask whether a node can be reinserted into the PQ during computation or not. In case of multi-label searches, labels do not change from inactive to active (at least in this thesis).

As a basic example of single-label search, Dijkstra's algorithm [33], which we recapitulate in Section 2.2.3 (see Algorithm 2.1), fits in the above taxonomy as follows: The label of a node $v \in V$ is a non-negative real number $d[v] \in \mathbb{R}_{\geq 0}$. The predecessor information of v is simply the node $u = p[v] \in V$ from that the label $d[v]$ of v has been improved last by relaxing an edge $u \to_c v \in E$. The key with respect to the PQ is simply the label $d[v]$ itself. Updating the label $d[v]$ is done by assigning $d[v] := \min\{d[v], d[u] + c\}$ when $u \to_c v$ is relaxed. The predecessor node $p[v]$ is updated by assigning $p[v] := u$ if relaxing $u \to_c v$ results in an improvement of the label $d[v]$ (i.e., if $d[u] + c < d[v]$ holds before relaxation). The initial node label of the start node s is simply $d[s] = 0$. As a result of the simple structure of the node labels, a node never gets active again once it changed from active to inactive. For this reason, one usually speaks of *settling* a node in the context of Dijkstra's algorithm.

The basic multi-label search [48, 60], which we recapitulate in Section 2.2.6 (see Algorithm 2.4), also fits in the above taxonomy. Because of the two-dimensional travel costs, there is no unique optimal path in general (see Section 2.1.3), and every tentative travel cost of a node corresponds to another tentative path. For this reason, travel costs and predecessor information of a node u are stored together in one label $(i, u, \gamma|\delta, i')$. Removing $(i, u, \gamma|\delta, i')$ from the PQ triggers the relaxation of the outgoing edges of u. Relaxing an edge $u \to_{c|d} v$ triggers the creation of a label $(j, v, \gamma + c|\delta + d, i)$ with j being the next available label id. A node label is never updated, but may be deleted if a newly created label of the same node dominates its two-dimensional cost. The predecessor information needs not to be updated separately of course. The start node s gets the single initial label $(0, s, 0|0, \perp)$ assigned. A label, once changed from active to inactive, can never get active again.

4.1.2 Predecessor Graphs

Dijkstra's algorithm [33] (see Section 2.2.3) not only computes shortest distances from a start node s to all other nodes in a graph, but also a shortest path tree. This shortest path tree is just the predecessor tree $G(p)$. For the other Dijkstra-like algorithms it is similar. For example, time-Dependent Dijkstra [35] (see Section 4.2.1), which computes $\mathrm{EA}_G(s, u, \tau_0)$ for all nodes $u \in V$ that are reachable from s with departure time τ_0, also builds a predecessor tree. This predecessor tree is an (s, τ_0)-EA-tree with route node s (see Section 3.1.3). For most Dijkstra-like algorithms in this thesis, however, the predecessor information must be able to encode multiple paths to every reachable node.

Single-Label Searches. In case of single-label searches, it can happen that not only a single departure time must be considered, but intervals of departure time (an example is TTP search [69], see Section 4.2.2). Other single-label searches compute approximate results (as in case of EA interval search, for example, see Section 4.2.5). In both cases, the predecessor information must be able to encode multiple paths to every reachable node. For this reason, it is no longer enough that the predecessor information $p[u]$ of a node u consists of a single node, as it is the case for Dijkstra's algorithm and time-dependent Dijkstra. Instead, $p[u]$ stores a *set* of predecessor nodes. Given a set of predecessor nodes $p[u]$ for all nodes $u \in V$ we often write $G(p) := (V(p), E(p))$ with

$$V(p) := \{u, v \in V \,|\, u \to v \in E(p)\}\ , \tag{4.1}$$

$$E(p) := \{u \to v \in E \,|\, u \in p[v]\} \tag{4.2}$$

to denote the *predecessor graph* of the underlying single-label search. This graph, which is implicitly represented by the predecessor information p, is simply the subgraph of G made up by all edges $u \to v$ with $u \in p[v]$.

Note that the earlier definition of $G(p)$ as predecessor tree (according to Equations (2.25) and (2.26)) perfectly fits into the above generalized definition of $G(p)$ as predecessor graph: If a node u is the predecessor of a node v (i.e., $p[v] = u$), we simply identify u with the singleton set $\{u\}$. If a node v has no predecessor (i.e. $p[v] = \perp$), we identify $\perp$ with the empty set $\emptyset$.

Multi-Label Searches. In case of multi-label searches, there is no separate set of predecessor nodes $p[u]$ for every node v. Instead, the predecessor information of u is attached to the labels $\ell \in L[u]$ of u (see Section 2.2.6 where a basic multi-label search is described). Every label $\ell = (i, u, \gamma | \delta, i')$ not only contains its own unique id i, but also the unique id i' of its preceding label. This leads to the definition of $G(L) := (V(L), E(L))$ with

$$V(L) := \{u, v \in V \,|\, u \to v \in E(L)\}\ , \tag{4.3}$$

$$E(L) := \{u \to v \in E \,|\, \text{there is } (i, v, \cdot|\cdot, i') \in L[v], (i', u, \cdot|\cdot, \cdot) \in L[u]\} \tag{4.4}$$

to denote the predecessor graph of the underlying multi-label computation.

4.2 Forward Single-Label Searches

This section describes basic time-dependent single-label search algorithms that run in forward direction. Single-label searches that run backward are described in Section 4.3. We start with two well-known time-dependent single-label search algorithms, namely *time-dependent Dijkstra* [35] (see Section 4.2.1) and *travel time profile (TTP) search* [69] (see Section 4.2.2). All other single label search algorithms considered in this section have been developed in the context of TCHs. These are *TTP interval search* (see Section 4.2.3), *approximate TTP search* (see Section 4.2.4), and *earliest arrival (EA) interval search* (see Section 4.2.5).

4.2.1 Time-Dependent Dijkstra

Given a start node s and a departure time τ_0, the well known *time-dependent Dijkstra* [35] (see Algorithm 4.1) computes $\mathrm{EA}_G(s,u,\tau_0)$ for all $u \in V$ as well as a corresponding (s,τ_0)-EA-tree. The initial node label $\tau[s]$ of the start node s is the departure time τ_0, the label $\tau[u]$ of a node u is the tentative EA time for traveling from s to u. The predecessor information $p[u]$ of u is the predecessor node of u on a corresponding tentative EA path. When we relax an edge $u \rightarrow_f v$, we update the label $\tau[v]$ of v by

$$\tau[v] := \min\{\tau[v], \mathbf{arr}\, f(\tau[u])\}\ . \tag{4.5}$$

Note that the arrival time function $\mathbf{arr}\, f$ is not directly present in G and cannot be evaluated directly. Instead, we evaluate the corresponding TTF f, which is stored with the edge $u \rightarrow_f v$. So, we actually compute $f(\tau[u]) + \tau[u]$ which is the same as $\mathbf{arr}\, f(\tau[u])$ (for efficient evaluation of TTFs see Section 3.1.4).

If $\tau[v]$ gets decreased (see Line 8 and 9), we also update the predecessor information by $p[v] := u$ (see Line 10). As PQ key of a node v we simply use its current label $\tau[v]$ (see Lines 11 and 12). Time-dependent Dijkstra works very similar to the original Dijkstra (see Algorithm 2.1 in Section 2.2.3). This implies that nodes are never reinserted into the PQ. The details are explained in the following.

Correctness. Step by step we prove that time-dependent Dijkstra really computes the desired result; that is, the EA times of all reachable nodes for a given

Algorithm 4.1. The time-dependent version of Dijkstra's algorithm computes final labels $\tau[u] = \mathrm{EA}_G(s,u,\tau_0)$ for all reachable nodes u (and $\tau[u] = \infty$ if u not reachable). The final predecessor graph $G(p)$, which is implicitly represented by the array p, is a (s,τ_0)-EA-tree.

```
1  procedure tdDijkstra($s : V,\ \tau_0 : \mathbb{R}$)
2      $\tau[u] := \infty$ for all $u \in V$, $\tau[s] := \tau_0$            // initial node labels
3      $p[u] := \perp$ for all $u \in V$                  // initial predecessor information
4      $Q := \{(s,\tau_0)\}$ : PriorityQueue
5      while $Q \neq \emptyset$ do
6          $u := Q.deleteMin()$
7          foreach $u \rightarrow_f v \in E$ do
8              if $\mathbf{arr}\, f(\tau[u]) \geq \tau[v]$ then continue
9              $\tau[v] := \mathbf{arr}\, f(\tau[u])$                 // update node label of v
10             $p[v] := u$                        // update predecessor information of v
11             if $v \notin Q$ then $Q.insert(v,\ \tau[v])$
12             else $Q.updateKey(v,\ \tau[v])$                     // maintain PQ
```

departure time as well as a corresponding EA tree. We start with the observation that every current label $\tau[v]$ of $v \in V$ is always the arrival time of the corresponding current path from s to v represented by the predecessor information. This implies that the labels maintained by time-dependent Dijkstra approach the EA times from above.

Lemma 4.1. *During execution of time-dependent Dijkstra, we always have* $\tau[v] = \mathbf{arr}\, f_{P_v}(\tau_0)$ *with* $P_v := \langle s = p[\ldots p[v] \ldots] \to \cdots \to p[v] \to v \rangle$ *for all* $v \in V$ *reached so far.*

Proof. We argue by induction over $|P_v|$. For $|P_v| = 1$ the statement is clearly true. For $|P_v| > 1$ assume $\tau[u] = \mathbf{arr}\, f_{P_u}(\tau_0)$ for all $P_u := \langle s = p[\ldots p[u] \ldots] \to \cdots \to p[u] \to u \rangle$ with $|P_u| < |P_v|$. As $\tau[v]$ has been set by relaxing the edge $p[v] \to_f v$, we have $\tau[v] = \mathbf{arr}\, f(\tau[p[v]]) = \mathbf{arr}\, f(\mathbf{arr}\, f_{P_{p[v]}}(\tau_0)) = \mathbf{arr}\, f_{P_v}(\tau_0)$. □

Corollary 4.2. *During execution of time-dependent Dijkstra,* $\tau[v] \geq \mathrm{EA}_G(s, v, \tau_0)$ *is always true for all* $v \in V$.

The following proof of correctness is very simple. It exploits that time-dependent Dijkstra relaxes the edges of at least one EA path from the start node s to each node u in the right order.

Lemma 4.3. *After time-dependent Dijkstra terminates,* $\tau[v] = \mathrm{EA}_G(s, v, \tau_0)$ *holds for all* $v \in V$.

Proof. Assume, there is a node v with $\tau[v] \neq \mathrm{EA}_G(s, v, \tau_0)$ after termination. Consider a prefix-optimal (s, v, τ_0)-EA-path $P = \langle s \to \cdots \to v \rangle$, which exists due to Lemma 3.10. Choose the minimum hop prefix path $\langle s \to \cdots \to u \to_f w \rangle$ of P such that $\tau[w] \neq \mathrm{EA}_G(s, w, \tau_0)$, which implies $\tau[u] = \mathrm{EA}_G(s, u, \tau_0)$. The edge $u \to_f w$ is relaxed as soon as u is removed from the PQ and we can be sure that

$$\tau[w] \leq \mathbf{arr}\, f(\tau[u]) = \mathbf{arr}\, f(\mathrm{EA}_G(s, u, \tau_0)) = \mathrm{EA}_G(s, w, \tau_0)$$

holds afterwards, because P is a prefix-optimal EA path. But Corollary 4.2 yields $\tau[w] \geq \mathrm{EA}_G(s, w, \tau_0)$ implying $\tau[w] = \mathrm{EA}_G(s, w, \tau_0)$, a contradiction. □

Together, Lemma 4.1 and 4.3 directly yield the correctness of time-dependent Dijkstra.

Corollary 4.4. *After time-dependent Dijkstra terminates,* $\tau[v] = \mathrm{EA}_G(s, t, \tau_0)$ *holds for all* $v \in V$, *and* $G(p)$ *is an* (s, τ_0)*-EA-tree.*

So far, we only showed that time-dependent Dijkstra yields an EA path for all reachable nodes after the PQ has run empty. But for us this is not enough.

To obtain fast route planning techniques, we want to terminate the execution of Dijkstra-like algorithms as early as possible. Typically, Dijkstra-like algorithms show a certain kind of monotonous behavior that enables us to stop their execution earlier.

Lemma 4.5. *Let $v_1, v_2, \dots$ be the order in that the nodes are removed from the PQ during time-dependent Dijkstra. Let $\tau_1, \tau_2, \dots$ be the respective node labels at time of removal. Then, $\tau_1 \le \tau_2 \le \dots$ holds.*

Proof. We argue by induction over the number i of removals from the PQ. For $i = 2$, consider the removal of v_2, which has been inserted by relaxing the edge $s \to_f v_2$ (with $s = v_1$). We have $\tau_2 = \mathbf{arr}\, f(\tau_1) = f(\tau_1) + \tau_1 \ge \tau_1$. For larger i, assume $\tau_1 \le \tau_2 \le \dots \le \tau_{i-1}$ and consider what happens when v_i is removed. The node v_i has been updated last by relaxing an edge $v_j \to_f v_i$ with $j < i$. So, we have $\tau_i = \mathbf{arr}\, f(\tau_j) = f(\tau_j) + \tau_j \ge \tau_j$. For $j = i - 1$, this means $\tau_i \ge \tau_{i-1}$. Otherwise, v_i is already in the PQ when v_{i-1} is removed, which implies $\tau_{i-1} \le \tau_i$. □

Interestingly, the above proof does not utilize the FIFO property. Instead, it is enough that the travel costs are non-negative.

Lemma 4.5 implies that $Q.min()$ never decreases during the execution of time-dependent Dijkstra. Moreover, it is obvious that time-dependent Dijkstra never increases a node label during execution. Both is used in the proof of Lemma 4.6 below. Note that all other Dijkstra-like algorithms in this thesis fulfill analogous properties.

Lemma 4.6. *The label $\tau[v]$ of v does not change anymore as soon as $Q.min() \ge \tau[v]$ is fulfilled. This implies that $\tau[v] = \mathrm{EA}_G(s, v, \tau_0)$ is guaranteed afterwards.*

Proof. As $Q.min()$ never decreases and $\tau[v]$ never increases, we know that the condition $Q.min() \ge \tau[v]$, once fulfilled, stays true until the execution ends. Let q_v and τ_v be the values of $Q.min()$ and $\tau[v]$ at the moment when $Q.min() \ge \tau[v]$ gets true (i.e., $q_v \ge \tau_v$). The label $\tau[v]$ can only be further changed by removing a node u from the PQ and then relaxing an edge $u \to_f v$. But relaxing this edge cannot further improve $\tau[v]$ because of $\mathbf{arr}\, f(\tau[u]) \ge \mathbf{arr}\, f(q_v) \ge \mathbf{arr}\, f(\tau_v) \ge \tau_v$. The reason is that $\tau[u] = Q.min() \ge q_v$ holds whenever a node u is removed after $Q.min() \ge \tau[v]$ got true. □

Obviously, the condition $Q.min() \ge \tau[v]$ gets true not later than the moment when v is removed from the PQ.

Corollary 4.7. *As soon as time-dependent Dijkstra removes a node v from the PQ, $\tau[v] = \mathrm{EA}_G(s, v, \tau_0)$ is always fulfilled afterwards. Time-dependent Dijkstra never reinserts a node into the PQ hence.*

It must be noted that other Dijkstra-like algorithm may reinsert nodes into the PQ. This is due to the usually more complicated structure of node labels. In the context of time-dependent Dijkstra, node labels are, in fact, very simple.

Of course, the condition $Q.min() \geq \tau[v]$ not only implies that the node label of v is the sought-after EA time of v. It also implies, that the predecessor information $p[v]$ represents a corresponding EA path to v, because of Lemma 4.1.

Corollary 4.8. *As soon as $Q.min() \geq \tau[v]$ gets true, $p[v]$ does not change anymore, and $P_v = \langle s = p[\dots p[v]\dots] \to \cdots \to p[v] \to v\rangle$ is guaranteed to be an (s, v, τ_0)-EA-path afterwards.*

We summarize Corollary 4.7 and 4.8 in a single statement.

Corollary 4.9. *As soon as time-dependent Dijkstra removes a node v from the PQ, we can be sure that $\tau[v] = \mathrm{EA}_G(s, v, \tau_0)$ holds, and that $\langle s = p[\dots p[v]\dots] \to \cdots \to p[v] \to v\rangle$ is a prefix-optimal (s, v, τ_0)-EA-path (if v is reachable from s).*

Running Time. Like in case of original Dijkstra, nodes are never reinserted into the PQ by time-dependent Dijkstra (see Corollary 4.7). As a consequence, time-dependent Dijkstra works quite similar as original Dijkstra: *deleteMin* is never invoked more than $|V|$ times, *insert* and *updateKey* cannot be invoked more than $O(|E|)$ times. The resulting running time is mainly governed by the performance of the PQ and by the performance of TTF evaluation (see Section 3.1.4 on page 90). So, assuming that road networks are practically sparse graphs (i.e., $|E| = \mathrm{O}(|V|)$) and that TTF evaluation needs average constant time (see Section 3.1.4), we expect that time-dependent Dijkstra has similar running time and memory usage as the original Dijkstra in practice; that is, $\mathrm{O}(|V| \cdot \log |V|)$ time if a binary heap is used to realize the PQ. Implementing TTF evaluation with binary search instead of the bucket structure described in Section 3.1.4 would result in a worst-case running time of $\mathrm{O}(|V| \cdot \log F + |V| \cdot \log |V|)$, with F being the maximum complexity of a TTF in G.

One-to-one Query. It is a well-known and obvious fact that time-dependent Dijkstra can be used to answer one-to-one EA queries, just like the original version of Dijkstra's algorithm can be used for one-to-one queries with constant travel costs. Given a start node s, a destination node t, and a departure time τ_0, one simply runs time-dependent Dijkstra. After termination, the path $\langle s = p[\dots p[u]\dots] \to \cdots \to p[t] \to t\rangle$ is an (s, t, τ_0)-EA-path and hence the desired result. However, it is not necessary to perform a full run of time-dependent Dijkstra. Instead, it can be stopped earlier. Due to Corollary 4.9, the computation can be stopped as soon as t is removed from the PQ. We refer to this modification as the *one-to-one version* of time-dependent Dijkstra.

4.2.2 Travel Time Profile Search

Given a start node s, *travel time profile search* [69] (or *TTP search* for short, see Algorithm 4.2) computes $\mathrm{TT}_G(s,u,\cdot)$ for all $u \in V$. Accordingly, the node label $F[u]$ of a node u is a tentative TTP for traveling from s to u. The tentative predecessor graph $G(p)$ contains the corresponding tentative EA paths found so far. When relaxing an edge $u \to_f v$ we update the label $F[v]$ of v by computing

$$F[v] := \min(F[v], f * F[u]) \ . \tag{4.6}$$

The PQ key of a node u is the minimum of its current label; that is, $\min F[u]$. The reader may notice that we check whether $F[v]$ lies completely under $g_{\text{new}} = f * F[u]$ and vice versa (see Lines 9 and 10). Both are essentially the same as computing $\min(g_{\text{new}}, F[v])$ and take $\mathrm{O}(|g_{\text{new}}| + |F[v]|)$ time hence. After termination, we have $F[u] = \mathrm{TT}_G(s,u,\cdot)$, and $G(p)$ contains an (s,u,τ)-EA-path for all $u \in V, \tau \in \mathbb{R}$ (if u is reachable from s).

Correctness. TTP search can be regarded as special case of approximate TTP search, whose final node labels $\big(\underline{F}[v], \overline{F}[v]\big)$ fulfill $\underline{F}[v](\tau) \le \mathrm{TT}_G(s,v,\tau) \le \overline{F}[v](\tau)$ for all $v \in V$ and all $\tau \in \mathbb{R}$ (see Section 4.2.4). So, a node label $F[v]$ of TTP search corresponds to a node label $\big(\underline{F}[v], \overline{F}[v]\big)$ of approximate TTP search with $\underline{F}[v] = \overline{F}[v]$. The following statements are essentially special cases of statements elaborately described a little later (see Section 4.2.4). So, proofs are omitted.

Algorithm 4.2. *Travel time profile (TTP) search* computes final labels $F[u] = \mathrm{TT}_G(s,u,\cdot)$ for all nodes u. The final predecessor graph $G(p)$ contains an (s,u,τ)-EA-path for all reachable nodes u and all departure times $\tau \in \mathbb{R}$.

```
1  procedure ttpSearch($s : V$)
2      $F[u] := \infty$ for all $u \in V$, $F[s] :\equiv 0$                // node labels are TTFs
3      $p[u] := \emptyset$ for all $u \in V$          // predecessor information consists of sets
4      $Q := \{(s,0)\}$ : PriorityQueue
5      while $Q \neq \emptyset$ do
6          $u := Q.deleteMin()$
7          foreach $u \to_f v \in E$ do
8              $g_{\text{new}} := f * F[u]$
9              if $g_{\text{new}}(\tau) \ge F[v](\tau)$ for all $\tau \in \mathbb{R}$ then continue
10             if $g_{\text{new}}(\tau) < F[v](\tau)$ for all $\tau \in \mathbb{R}$ then $p[v] := \emptyset$
11             $p[v] := \{u\} \cup p[v]$   // remove or remember tentative predecessors of v
12             $F[v] := \min(F[v], g_{\text{new}})$             // update tentative node label of v
13             if $v \notin Q$ then $Q.insert(v, \min F[v])$
14             else $Q.updateKey(v, \min F[v])$              // minimum of TTF is PQ key
```

Like time-dependent Dijkstra, TTP search shows a typical monotonous behavior. Nearly all Dijkstra-like algorithms considered in this thesis do this.

Lemma 4.10. *Let $v_1, v_2, \dots$ be the order in that the nodes are removed from the PQ during TTP search. Let $F_1, F_2, \dots$ be the respective node labels at time of removal. Then,* $\min F_1 \leq \min F_2 \leq \dots$ *holds.*

So, $Q.min()$ never decreases during TTP search runs. Obviously, $\max F[v]$ never increases for any node $v \in V$. Together this yields a criterion, when node labels and predecessor information get final for a node v.

Lemma 4.11. *As soon as $Q.min() \geq \max F[v]$ becomes true for a node v, $F[v]$ and $p[v]$ do not change anymore. From that moment on, $F[v] = \mathrm{TT}_G(s,v,\cdot)$ holds true and $G(p)$ is guaranteed to contain a prefix-optimal (s,v,τ)-EA-path for all $\tau \in \mathbb{R}$ (provided that v is reachable from s).*

Lemma 4.12. *After TTP search terminates, $F[v] = \mathrm{TT}_G(s,v,\cdot)$ holds for all $v \in V$. Also, the predecessor graph $G(p)$ contains a prefix-optimal (s,v,τ)-EA-path in G for all $\tau \in \mathbb{R}$ if v is reachable from s.*

Running Time. In contrast to time-dependent Dijkstra, TTP search may reinsert nodes into the PQ after they have been removed. In our experience, however, this happens rarely and has not much impact on the performance in the context of time-dependent road networks. Nevertheless, TTP search is very expensive in time and space and not feasible for larger road networks. This is confirmed by our experiments (see Section 5.6). The reason is that the complexity of the node labels tends to increase as TTP search goes on.

To understand that, consider how complex $g * f$ and $\min(f,g)$ can get with $f,g \in \mathscr{F}_\Pi$, and how this involves that node labels get more and more complex during TTP search. In the worst case, we have $|g * f| = |f| + |g|$ and $|\min(f,g)| = \Omega(|f| + |g|)$ (see Section 3.1.4). First, we take a closer look at $|g * f|$. In practice, we rather expect $|g * f| \approx |f| + |g|$ than $|g * f|$ being small. This is because it is not probable that many of the values

$$\dots, f_0.x + f_0.y,\, f_1.x + f_1.y, \dots\; f_{|f|-1}.x + f_{|f|-1}.y, \dots$$

coincide with many of the x-values $\dots, g_0.x, g_1.x, \dots, g_{|g|-1}.x, \dots$. It also seems not so probable that many bend points of $g * f$ are redundant in the sense of being collinear with neighboring bend points. It seems likely, hence, that most bend points of f and g have a corresponding bend point in $g * f$.

In case of $\min(f,g)$, things are different. Here, it is possible that TTFs may not intersect very often. If this is the case, we rather expect that $|\min(f,g)|$ stays near $|f|$ or $|g|$. If f and g have similar complexity (i.e., $|f| \approx |g|$), we optimistically assume $|\min(f,g)| \approx |f| \approx |g|$. However, $|g * f| \approx |f| + |g|$ is bad enough in practice, even if $|\min(f,g)|$ tends to be well-behaved (i.e., $|\min(f,g)| \approx |f| \approx |g|$ instead of $|\min(f,g)| = \Omega(|f|+|g|)$). To understand that, consider the relaxation of an edge $u \rightarrow_f v$. The tentative TTP $F[u]$ is linked with the TTF f and we basically expect $|f * F[u]| \approx |F[u]| + |f|$, as explained above. In other words, $|f * F[u]| > |F[u]|$ is not unlikely. As $f * F[u]$ is used to update the node label of v by assigning $F[v] := \min(F[v], f * F[u])$, we expect that the complexity of the updated label tends to be larger than $|F[u]|$; that is,

$$\left|\min(f * F[u], F[v])\right| \approx |f| + \left|F[u]\right| \gtrapprox \left|F[v]\right| ,$$

even if the minimum does not contribute to the increase. So, the complexity of the processed node labels should increase during TTP search. We expect that the processing of node labels is very time-consuming hence.

Assume, as a simple example, that G is a path $\langle u_1 \rightarrow_{f_1} \cdots \rightarrow_{f_{\ell-1}} u_\ell \rangle$ with $|f_i| = M$ for all $i \in \{1,\ldots,\ell-1\}$. Now, run TTP search with start node $s := u_1$. Relaxing the edges $u_1 \rightarrow_{f_1} u_2, u_2 \rightarrow_{f_2} u_3, \ldots, u_{\ell-1} \rightarrow_{f_{\ell-1}} u_\ell$ one after another results in the node labels $F[u_2], F[u_3], \ldots, F[u_\ell]$, respectively, with worst-case complexity

$$\left|F[u_i]\right| = (i-1)M ,$$

because of $F[u_i] = f_{i-1} * \cdots * f_1$. Then, we expect that relaxing the i-th edge $u_i \rightarrow u_{i+1}$ takes $\Theta(iM)$ time. This leads to an overall running time of

$$\Theta\Big(\sum_{i=1}^{\ell-1} iM\Big) = \Theta(\ell^2 M) \tag{4.7}$$

in the worst case. If one chooses the x-values of f_i in a way that none of them coincides with the y-values of $\mathbf{arr}(f_{i-1} * f_{i-2} * \cdots * f_1)$, then this worst-case really occurs. Note that the x-values of f_i can also be chosen in a way that $f_i * f_{i-1} * \cdots * f_1$ does not contain any neighboring collinear bend points.

One-to-one Query. Analogous to time-dependent Dijkstra, TTP search can be used to answer one-to-one TTP queries. To compute $\mathrm{TT}_G(s,t,\cdot)$ for given $s,t \in V$, one simply runs TTP search returning the TTP $F[t]$ after termination. It is not necessary to run TTP search completely however. Instead the search can be stopped as soon as $Q.min() \geq \max F[t]$ gets true. Then, $F[t]$ has already reached its final value $\mathrm{TT}_G(s,t,\cdot)$ and $G(p)$ already contains a prefix-optimal (s,t,τ)-EA-path for all $\tau \in \mathbb{R}$. This follows from Corollary 4.11.

4.2.3 Travel Time Profile Interval Search

As TTP search (see Algorithm 4.2 in Section 4.2.2) is so expensive, we use *travel time profile interval search* (or *TTP interval search* for short, see Algorithm 4.3) as

Algorithm 4.3. The *travel time profile (TTP) interval search* computes final labels $[q[u], r[u]] = \left[\underline{\mathrm{TT}}_G(s,u), \overline{\mathrm{TT}}_G(s,u)\right]$ for all nodes u. As in case of TTP search, $G(p)$ contains an (s,u,τ)-EA-path for all reachable nodes u and all departure times $\tau \in \mathbb{R}$ after termination.

```
procedure ttpIntervalSearch($s : V$)
    $[q[u], r[u]] := [\infty, \infty]$ for all $u \in V$, $[q[s], r[s]] := [0, 0]$        // labels are intervals
    $p[u] := \emptyset$ for all $u \in V$                    // predecessor information consists of sets
    $Q := \{(s, 0)\} : PriorityQueue$
    while $Q \neq \emptyset$ do
        $u := Q.deleteMin()$
        foreach $u \to_f v \in E$ do
            $[q_{\text{new}}, r_{\text{new}}] := \left[q[u] + \min f,\ r[u] + \max f\right]$
            if $q_{\text{new}} > r[v]$ then continue
            if $r_{\text{new}} < q[v]$ then $p[v] := \emptyset$          // remove suboptimal predecessors
            $p[v] := \{u\} \cup p[v]$                   // remember a tentative predecessor of v
            if $q_{\text{new}} \geq q[v]$ and $r_{\text{new}} \geq r[v]$ then continue
            $[q[v], r[v]] := \left[\min\{q[v], q_{\text{new}}\},\ \min\{r[v], r_{\text{new}}\}\right]$     // update label of v
            if $v \notin Q$ then $Q.insert(v, q[v])$
            else $Q.updateKey(v, q[v])$        // lower bound of the interval is PQ key
```

a relatively loose approximation of TTP search. Instead of $\mathrm{TT}_G(s,u,\cdot)$ it computes the interval $[\underline{\mathrm{TT}}_G(s,u), \overline{\mathrm{TT}}_G(s,u)]$ for every node $u \in V$. Hence, the label of a node u is no more a tentative TTP but a tentative interval $[q[u], r[u]]$. The tentative predecessor graph $G(p)$ contains corresponding tentative EA paths for all possible departure times $\tau \in \mathbb{R}$. When relaxing an edge $u \to_f v$ we update the label of a node v by

$$[q[v], r[v]] := \left[\min\{q[v],\, q[u] + \min f\}, \min\{r[v],\, r[u] + \max f\}\right] . \tag{4.8}$$

As PQ key of a node u we use $q[u]$. Like in case of TTP search, nodes may also be reinserted into the PQ from time to time. Again, this happens rarely in practice.

Note that the predecessor graph $G(p)$ computed by TTP interval search is expected to contain more edges than the one computed by exact TTP search (see Section 4.2.2) and also more edges than the one computed by approximate TTP search (see Section 4.2.4).

Correctness. We argue that TTP interval search is a special case of approximate TTP search (see Section 4.2.4). This is analogous to TTP search (see Section 4.2.2), which is a special case of approximate TTP search, too. In case of TTP interval search, the lower and upper bounds $\underline{F}$ and $\overline{F}$, respectively, are simply constant functions. Most of the following statements are essentially special cases of statements elaborately described a little later (see Section 4.2.4). So, proofs are omitted.

Lemma 4.13. *Let $v_1, v_2, \dots$ be the order in that the nodes are removed from the PQ during TTP interval search. Let $[q_1, r_1], [q_2, r_2], \dots$ be the respective node labels at time of removal. Then, $q_1 \le q_2 \le \dots$ holds.*

Lemma 4.14. *As soon as $Q.min() > r[v]$ becomes true for a node v, $[q[v], r[v]]$ and $p[v]$ do not change anymore. From that moment on, $\mathrm{TT}_G(s,v,\tau) \in [q[v], r[v]]$ is guaranteed for all $\tau \in \mathbb{R}$. Also, $G(p)$ contains a prefix-optimal (s,v,τ)-EA-path for all $\tau \in \mathbb{R}$ ever after.*

Note that the final label $[q[v], r[v]]$ of a node v is not only some interval containing $\mathrm{TT}_G(s,v,\tau)$ for all $\tau \in V$. In fact, it is the smallest possible interval that can be computed in terms of minima and maxima of TTFs.

Lemma 4.15. *After TTP interval search terminates, we know that $[q[v], r[v]] = \big[\underline{\mathrm{TT}}_G(s,v), \overline{\mathrm{TT}}_G(s,v)\big]$ holds for all $v \in V$.*

Proof. Consider a path $\langle s = v_1 \to_{f_1} \cdots \to v_{k-1} \to_{f_{k-1}} v_k \rangle$ with

$$\underline{\mathrm{TT}}_G(s,v_j) = \sum_{i=1}^{j-1} \min f_i$$

for $1 \le j \le k$ (such a path is simply a shortest path with respect to constant edge costs that we obtain by replacing each edges TTF g with $\min g$). Choose j minimal such that $q[v_j] = \underline{\mathrm{TT}}_G(s,v_j)$ but $q[v_{j+1}] \ne \underline{\mathrm{TT}}_G(s,v_{j+1})$. This yields

$$\begin{aligned}\underline{\mathrm{TT}}_G(s,v_{j+1}) &= \underline{\mathrm{TT}}_G(s,v_j) + \min f_j \\ &= q[v_j] + \min f_j \ge q[v_{j+1}] > \underline{\mathrm{TT}}_G(s,v_{j+1})\ ,\end{aligned}$$

which cannot be the case. Regarding the upper bound $r[v]$, one can argue analogously. □

Lemma 4.15 enables us to formulate the statement of Lemma 4.14 in a more specific way.

Corollary 4.16. *As soon as $Q.min() > r[v]$ becomes true for a node v,*

$$\mathrm{TT}_G(s,v,\tau) \in [q[v], r[v]] = \big[\underline{\mathrm{TT}}_G(s,v), \overline{\mathrm{TT}}_G(s,v)\big]$$

is guaranteed for all $\tau \in \mathbb{R}$. Also, $G(p)$ contains a prefix-optimal (s,v,τ)-EA-path for all $\tau \in \mathbb{R}$ ever after.

Running Time. To our experience, TTP interval search has similar running time as time-dependent Dijkstra (see Section 4.2.1) on road networks. On the one hand, this is because reinsertions into the PQ do not happen too often. On the other hand, this comes from the fact that edge relaxations take similar running time as in case of time-dependent Dijkstra. To do so, we store $\min f$ and $\max f$ with each edge $u \to_f v$ and perform edge relaxations in constant time then. Expecting only few reinserts into the PQ, we expect a running time near $\mathrm{O}(|V| \cdot \log |V|)$ hence.

4.2.4 Approximate Travel Time Profile Search

Approximate travel time profile search (or *approximate TTP search* for short, see Algorithm 4.4) is, like TTP interval search, an approximate version of TTP search. Approximate TTP search is far more accurate than TTP interval search but runs significantly slower. However, it still runs much faster than TTP search. The idea is to use approximate TTFs and TTPs instead of exact ones, because they tend to have much less bend points and can be processed much faster hence.

For all nodes $u \in V$ the approximate TTP search computes pairs of TTFs $(\underline{F}[u], \overline{F}[u])$ that fulfill $\underline{F}[u](\tau) \leq \mathrm{TT}_G(s,u,\tau) \leq \overline{F}[u](\tau)$ for all $\tau \in \mathbb{R}$ after termination. We call $\underline{F}[u]$ and $\overline{F}[u]$ *lower* and *upper bounds* respectively. Correspondingly, the pairs of TTFs $(\underline{F}[u], \overline{F}[u])$ act as node labels. They represent tentative lower and upper bounds of the TTP for traveling from s to u during computation. The tentative predecessor graph $G(p)$ contains corresponding tentative EA paths for all departure times $\tau \in \mathbb{R}$. As PQ key of a node u we use $\min \underline{F}[u]$. Just like in case of TTP search, reinserts into the PQ are possible.

When an edge $u \to_f v$ is relaxed (see Line 15), the label $(\underline{F}[v], \overline{F}[v])$ of v is updated by

$$\big(\underline{F}[v], \overline{F}[v]\big) := \Big(\min\big(\underline{F}[v], \underline{f} * \underline{F}[u]\big), \min\big(\overline{F}[v], \overline{f} * \overline{F}[u]\big)\Big) , \tag{4.9}$$

where $\underline{f}$ and $\overline{f}$ are lower and upper bounds of f, respectively (i.e., $\underline{f}(\tau) \leq f(\tau) \leq \overline{f}(\tau)$ for all $\tau \in \mathbb{R}$). Figure 4.1 illustrates this situation. The bounds $\underline{f}$ and $\overline{f}$ (see Line 8 and 9) can either be computed during the execution of the algorithm or during a preprocessing step. The use of $\underline{f}$ and $\overline{f}$ instead of f already makes $\underline{F}$ and $\overline{F}$ lower and upper bounds respectively. The running time can be further reduced, however, if $\underline{F}$ and $\overline{F}$ are further approximated. But this must be done during execution (see Lines 16 to19). Note that the predecessor graph $G(p)$ computed by approximate TTP search is expected to contain more edges than the one computed by exact TTP search (see Section 4.2.2), but less edges than the one computed by TTP interval search (see Section 4.2.3).

Algorithm 4.4. Given an $s \in V$ the *approximate travel time profile (TTP) search* computes pairs of bounding TTFs $\left(\underline{F}[u], \overline{F}[u]\right)$; that is, $\underline{F}[u](\tau) \leq \mathrm{TT}_G(s,u,\tau) \leq \overline{F}[u](\tau)$ holds for all $u \in V$ and $\tau \in \mathbb{R}$. After termination $G(p) \subseteq G$ contains a prefix-optimal (s,u,τ)-EA-path for all nodes u that are reachable from s and for all $\tau \in \mathbb{R}$.

```
procedure $approximateTtpSearch(s : V)$
    $\underline{F}[u] := \overline{F}[u] := \infty$ for all $u \in V$, $\underline{F}[s] := \overline{F}[s] :\equiv 0$
    $p[u] := \emptyset$ for all $u \in V$
    $Q := \{(s, 0)\} : PriorityQueue$
    while $Q \neq \emptyset$ do
        $u := Q.deleteMin()$
        foreach $u \to_f v \in E$ do
            let $\overline{f} \in \mathscr{F}_\Pi$ be an upper bound of $f$ with $|\overline{f}| \ll |f|$
            let $\underline{f} \in \mathscr{F}_\Pi$ be a lower bound of $f$ with $|\underline{f}| \ll |f|$
            $(\underline{g}_{\text{new}}, \overline{g}_{\text{new}}) := (\underline{f} * \underline{F}[u], \overline{f} * \overline{F}[u])$
            if $\underline{g}_{\text{new}}(\tau) \geq \overline{F}[v](\tau)$ for all $\tau \in \mathbb{R}$ then continue
            if $\overline{g}_{\text{new}}(\tau) < \underline{F}[v](\tau)$ for all $\tau \in \mathbb{R}$ then $p[v] := \emptyset$
            $p[v] := \{u\} \cup p[v]$
            if $\underline{g}_{\text{new}}(\tau) \geq \underline{F}[v](\tau) \wedge \overline{g}_{\text{new}}(\tau) \geq \overline{F}[v](\tau)$ for all $\tau \in \mathbb{R}$ then continue
            $(\underline{F}[v], \overline{F}[v]) := \left(\min(\underline{F}[v], \underline{g}_{\text{new}}), \min(\overline{F}[v], \overline{g}_{\text{new}})\right)$
            if $|\overline{F}[v]|$ gets too large then
                replace $\overline{F}[v]$ by an upper bound of $\overline{F}[v]$ with less bend points
            if $|\underline{F}[v]|$ gets too large then
                replace $\underline{F}[v]$ by a lower bound of $\underline{F}[v]$ with less bend points
            if $v \notin Q$ then $Q.insert(v, \min \underline{F}[v])$
            else $Q.updateKey(v, \min \underline{F}[v])$
```

Computing Lower and Upper Bounds. The idea behind the approximation is to save running time by reducing the complexity of the involved TTFs. This means we want to compute lower and upper bounds $\underline{f}$, $\overline{f}$, $\underline{F}[v]$, and $\overline{F}[v]$ with a preferably small number of bend points. To generate such lower and upper bounds of a TTF f with possibly small complexity and good accuracy, we choose a small relative error $\varepsilon > 0$. Then, we compute an upper bound $\overline{f} \in \mathscr{F}_\Pi$ of f with maximum relative error ε; that is, $\overline{f}$ fulfills $f(\tau) \leq \overline{f}(\tau) \leq (1+\varepsilon)f(\tau)$ for all $\tau \in \mathbb{R}$. To do so, one may want to use a well-known algorithm which has been discovered independently by Rama [72] as well as Douglas and Peucker [34]. At least for the concave parts of f this approach has obvious problems. If concave parts are rather rare, however, this may still work well. But we have not tried this and do

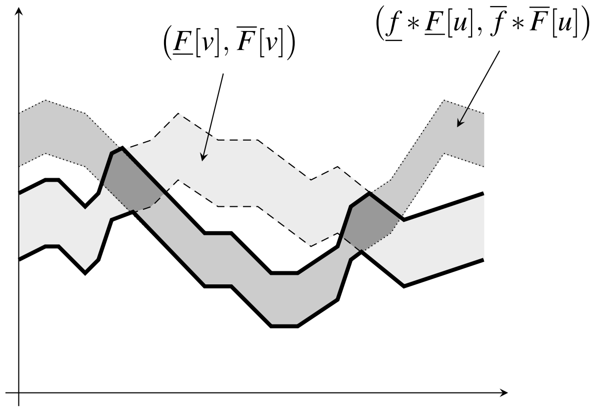

$(\underline{F}[v], \overline{F}[v])$
$(\underline{f} * \underline{F}[u], \overline{f} * \overline{F}[u])$

Figure 4.1. Illustration of $\big(\min(\underline{F}[v], \underline{f} * \underline{F}[u]), \min(\overline{F}[v], \overline{f} * \overline{F}[u])\big)$ as computed by approximate TTP search when the label of v is updated during relaxation of an edge $u \to_f v$ with lower bound $\underline{f}$ and upper bound $\overline{f}$. The thick lines denote $\min(\underline{F}[v], \underline{f} * \underline{F}[u])$ and $\min\big(\overline{F}[v], \overline{f} * \overline{F}[u]\big)$ each. Note that the exact TTFs, though they may be not known, lie somewhere in their respective "channel".

not know how this performs in practice.

Instead, we use the less popular Imai-Iri-algorithm [52]. It has double advantage: First, it runs in $\mathrm{O}(|f|)$ time. Second, the resulting approximate TTF $\overline{f}$ has minimal possible complexity $\big|\overline{f}\big|$ such that $f(\tau) \le \overline{f}(\tau) \le (1+\varepsilon)f(\tau)$ holds for all $\tau \in \mathbb{R}$. The Imai-Iri-algorithm computes a polyline with a minimal number of bend points that lies in the "channel" formed by the graphs of f and $(1+\varepsilon)f$ (see Figure 4.2 for an illustration). A lower bound could be computed in an analogous manner, also using the Imai-Iri-algorithm.

However, a lower bound is already at hand, namely the lower bound $\underline{f} := (1+\varepsilon)^{-1}\overline{f}$ which is implicitly defined by $\overline{f}$. This lower bound $\underline{f}$ fulfills $(1+\varepsilon)^{-1}f(\tau) \le \underline{f}(\tau) \le f(\tau)$ for all $\tau \in \mathbb{R}$. It should be noted that the Imai-Iri-algorithm is relatively complicated and not easy to implement. In our experiments (see Section 5.6), we use an implementation provided by Neubauer [66].

Running Time Versus Accuracy. The reason, why approximate TTP search runs usually much faster than TTP search, is that linking and minimum of TTFs takes less time if the complexity of the linked TTFs is low. This is just what we accomplish by using lower and upper bounds instead of exact TTFs. Using lower and upper bounds that have small error but much less bend points than exact TTFs, can reduce running time, while accuracy stays reasonable. There is an obvious tradeoff between running time and accuracy. Allowing larger relative errors on the computation of lower and upper bounds usually saves bend points. This results in better running times but worse accuracy. For smaller relative errors

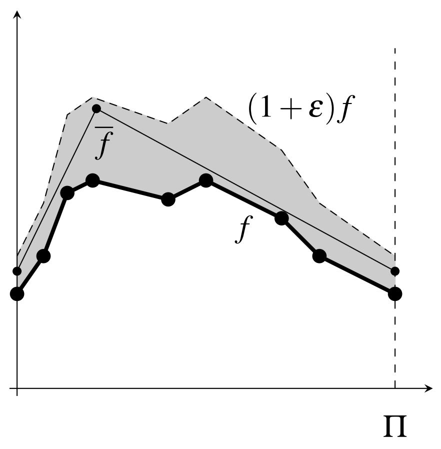


Figure 4.2. An upper bounds $\overline{f}$ (thin) of a TTF $f \in \mathscr{F}_\Pi$ (thick) with relative error not greater than $\varepsilon = 0.4$. The upper bound $\overline{f}$ lies in the "channel" (gray) formed by the graph of f and and the graph of $(1+\varepsilon)f$ (dashed). With only two bend points, $\overline{f}$ has significantly less bend points than f, which has eight bend points in $[0, \Pi)$.

it is the other way round with worse running times but better accuracy.

Note that the complexity of the node labels has more impact on the running time than the complexity of the lower and upper bounds of TTFs f with $u \rightarrow_f v$. To explain that, we return to the simple example from Section 4.2.2 (see page 125). Again, we assume that G is simply a path $\langle u_1 \rightarrow_{f_1} \cdots \rightarrow_{f_{\ell-1}} u_\ell \rangle$ with $|f_i| = M$ for all $i \in \{1, \dots, \ell-1\}$. Running a TTP search on G starting from $s = u_1$ takes up to $\Theta\big(M + 2M + \cdots + (\ell-1)M\big) = \Theta(\ell^2 M)$ time. Now, we run approximate TTP search instead of TTP search.

First, assume that we do not reduce the complexity of the node labels (i.e., of $\underline{F}[v]$ and $\overline{F}[v]$). Instead, we only use precomputed lower and upper bounds $\underline{f_i}$ and $\overline{f_i}$ for all $i \in \{1, \dots, \ell-1\}$ respectively. Assume, that the complexity of all these TTFs is reduced by a factor of $\alpha > 1$; that is,

$$|\underline{f_1}|, \dots, |\underline{f_{\ell-1}}|, |\overline{f_1}|, \dots, |\overline{f_{\ell-1}}| \leq \frac{M}{\alpha} .$$

As a consequence, we expect that the running time is reduced by a factor of α, too. But ℓ contributes quadratically to the running time. So, reducing the complexity of $f_1, \dots, f_{\ell-1}$ only helps if ℓ is relatively small.

Second, assume that we do not use lower and upper bounds of f when relaxing an edge $u \rightarrow_f v$, but that we reduce the complexity of $\underline{F}[v]$ and $\overline{F}[v]$ whenever $\big|\underline{F}[v]\big|$ and $\big|\overline{F}[v]\big|$ get too large (see Lines 16–19). So, if all TTFs do not vary too much, we can always expect that a maximum complexity N exists such that $\big|\underline{F}[v]\big|, \big|\overline{F}[v]\big| \leq N$ holds before v is inserted into the PQ. Then, the running time lies in

$$\mathrm{O}\Big(\sum_{i=1}^{\ell-1}(N+M)\Big)=\mathrm{O}\big(\ell(N+M)\big)\ .$$

This means we get rid off the quadratic running time by approximating the node labels if they get too complex. Note that approximating a node label $\big(\underline{F}[v],\overline{F}[v]\big)$ only takes $\mathrm{O}\big(\big|\underline{F}[v]\big|+\big|\overline{F}[v]\big|\big)=\mathrm{O}(N+M)$, time if the Imai-Iri algorithm is used.

To summarize, reducing the complexity of the node labels is much more effective than using lower and upper bounds during edge relaxation. However, only using lower and upper bounds $\underline{f}$ and $\overline{f}$ during the relaxation of an edge $u \to_f v$ is nevertheless interesting; namely,

- if the paths in the graph do not have many hops (i.e., ℓ is not large),
- if $\big|\underline{f}\big|,\big|\overline{f}\big| \ll |f|$ holds for most such TTFs f, and
- if all such lower and upper bounds $\underline{f}$ and $\overline{f}$ are precomputed.

In case of ATCHs, a variant of TCHs that enables fast, exact, and space efficient route planning (see Section 5.4), all three conditions are fulfilled. There is also an inexact variant of TCHs exploiting this to provide even faster but inexact TTP queries (see Section 5.5).

Correctness. We now discuss the correctness of approximate TTP search (see Algorithm 4.4). This means we examine whether it really computes the desired lower and upper bounds and whether its predecessor graph contains EA paths to every reachable node for all departure times. Note that approximate TTP search includes TTP search (see Algorithm 4.2 in Section 4.2.2) and TTP interval search (see Algorithm 4.3 in Section 4.2.3) as special cases. The proofs described here can also be applied to those algorithms. We start with a very basic observation.

Lemma 4.17. *Consider the TTFs $f,g \in \mathscr{F}_\Pi$ and the corresponding lower and upper bounds $\underline{f},\overline{f},\underline{g},\overline{g}$. Then, the following two statements hold true:*

1. $\min(\underline{f},\underline{g})$ *is a lower and* $\min(\overline{f},\overline{g})$ *an upper bounds of* $\min(f,g)$*.*
2. $\underline{g}*\underline{f}$ *are lower and* $\overline{g}*\overline{f}$ *are upper bounds of* $g*f$*.*

Proof. The first statement is clearly true. Regarding the second one, we only proof that $\overline{g}*\overline{f}$ is an upper bound of $g*f$. For lower bounds everything works analogously. Because $\mathbf{arr}\,\overline{f}$ and $\mathbf{arr}\,\overline{g}$ are upper bounds of $\mathbf{arr}\,f$ and $\mathbf{arr}\,g$, respectively, we calculate

$$\mathbf{arr}\,\overline{g}\big(\mathbf{arr}\,\overline{f}(\tau)\big)\geq\mathbf{arr}\,g\big(\mathbf{arr}\,\overline{f}(\tau)\big)\geq\mathbf{arr}\,g(\mathbf{arr}\,f(\tau))\ ,$$

utilizing $g*f(\tau)=\mathbf{arr}\,g(\mathbf{arr}\,f(\tau))-\tau$ and that g fulfills the FIFO property, which means that $\mathbf{arr}\,g$ is increasing. □

Note that the relative error of $\overline{g} * \overline{f}$ can be greater than the relative errors of $\overline{f}$ and $\overline{g}$. This is because the maximum possible relative error also depends on the maximum slope of $\overline{g}$ [42]. Similarly, the relative error of $\underline{g} * \underline{f}$ depends on the maximum slope of $\underline{g}$.

Just like the other Dijkstra-like algorithms considered in this thesis, approximate TTP search shows a typical monotonous behavior.

Lemma 4.18. *Let $v_1, v_2, \dots$ be the order in that the nodes are removed from the PQ during approximate TTP search (nodes may be removed multiple times). Let $\left(\underline{F_1}, \overline{F_1}\right), \left(\underline{F_2}, \overline{F_2}\right), \dots$ be the respective node labels at the time when each node is removed. Then, $\min \underline{F_1} \leq \min \underline{F_2} \leq \dots$ holds.*

Proof. One can argue by induction over the number of removals from the PQ, analogous to the proof of Lemma 4.5. □

Like in case of time-dependent Dijkstra, we find that tentative node labels and tentative predecessor information always correspond to each other, though the situation is more complicated here.

Lemma 4.19. *While approximate TTP search runs, $G(p)$ contains*

1. *a path $P_{v,\tau} = \langle s \to \cdots \to v \rangle$ with $\underline{F}[v](\tau) \leq f_{P_{v,\tau}}(\tau) \leq \overline{F}[v](\tau)$ for all reached nodes v and all $\tau \in \mathbb{R}$,*
2. *all such paths $P_{v,\tau}$ where all prefixes $\langle s \to \cdots \to u \rangle \subseteq P_{v,\tau}$ (including $P_{v,\tau}$ itself) additionally fulfill $f_{\langle s \to \cdots \to u \rangle}(\tau) < \min\left\{\overline{F}[u](\tau), Q.min()\right\}$, and*
3. *no path $R = \langle s \to \cdots \to v \rangle$ with $f_R(\tau) < \underline{F}[v](\tau)$ for any $\tau \in \mathbb{R}$.*

Proof. We start with proving that $P_{v,\tau} \subseteq G(p)$ exists for all reached nodes $v \in V$ and all $\tau \in \mathbb{R}$ such that $f_{P_{v,\tau}}(\tau) \leq \overline{F}[v](\tau)$ holds.

Assume, there is a reached node $v_0 \in V$ such that $G(p)$ contains no path from s to v_0 at all. Presume, w.l.o.g., that v_0 has no incoming edge in $G(p)$. Consider the first incoming edge $u \to_f v_0$ of v_0 in G that is relaxed by the approximate TTP search. Surely, u is added to $p[v_0]$. Moreover, $p[v_0]$ is never emptied without adding another node. So, v_0 has at least one incoming edge in $G(p)$—a contradiction.

Further, assume there is $v_0 \in V$ and $\tau_0 \in \mathbb{R}$ such that $f_{\langle s \to \cdots \to v_0 \rangle}(\tau_0) > \overline{F}[v_0](\tau_0)$ holds for all paths $\langle s \to \cdots \to v_0 \rangle \subseteq G(p)$. We set $w := v_0$. There must be an edge $u \to_f w$ in $G(p)$ with $\overline{F}[w](\tau_0) = \overline{f} * \overline{F}[u](\tau_0) \geq f * \overline{F}[u](\tau_0)$, which has been relaxed earlier. If all paths $\langle s \to \cdots \to u \rangle \subseteq G(p)$ fulfill $f_{\langle s \to \cdots \to u \rangle}(\tau_0) > \overline{F}[u](\tau_0)$, we set $w := u$, reapply this argument, and repeat this until some path $\langle s \to \cdots \to u \rangle \subseteq G(p)$ fulfills $f_{\langle s \to \cdots \to u \rangle}(\tau_0) \leq \overline{F}[u](\tau_0)$. Both $f_{\langle s \to \cdots \to u \to_f w \rangle}(\tau_0) > \overline{F}[w](\tau_0)$

and $\overline{F}[w](\tau_0) \geq f * \overline{F}[u](\tau_0)$ also hold true. So, to obtain a contradiction, we argue

$$\begin{aligned} f_{\langle s \to \cdots \to u \to_f w \rangle}(\tau_0) &> \overline{F}[w](\tau_0) \geq f * \overline{F}[u](\tau_0) \\ &\geq f * f_{\langle s \to \cdots \to u \rangle}(\tau_0) = f_{\langle s \to \cdots \to u \to_f w \rangle}(\tau_0) \ , \end{aligned}$$

utilizing Equation (3.13) and that f fulfills the FIFO property.

Next, we show that all paths $P := \langle s \to \cdots \to v \rangle \subseteq G(p)$ fulfill $f_P(\tau) \geq \underline{F}[v](\tau)$ for all $\tau \in \mathbb{R}$. So, assume $P_0 := \langle s \to \cdots \to u \to_f v_0 \rangle \subseteq G(p)$ with $f_{P_0}(\tau_0) < \underline{F}[v_0](\tau_0)$ for some $\tau_0 \in \mathbb{R}$. W.l.o.g., presume $f_{\langle s \to \cdots \to u \rangle}(\tau_0) \geq \underline{F}[u](\tau_0)$. The edge $u \to_f v_0$ lies in $G(p)$ and has already been relaxed hence. To obtain a contradiction, we utilize Equation (3.13) as well as (3.14) and argue

$$f_{P_0}(\tau_0) < \underline{F}[v_0](\tau_0) \leq f * \underline{F}[u](\tau_0) \leq f * f_{\langle s \to \cdots \to u \rangle}(\tau_0) = f_{P_0}(\tau_0) \ .$$

At this point, we know that the first and the third statement must both be true. To show the second statement, assume a path $\langle s \to \cdots \to u \to_f v_0 \rangle \not\subseteq G(p)$ such that all its prefixes $\langle s \to \cdots \to u' \rangle \subseteq \langle s \to \cdots \to u \to_f v_0 \rangle$ fulfill

$$f_{\langle s \to \cdots \to u' \rangle}(\tau_0) < \min\left\{\overline{F}[u'](\tau_0), Q.min()\right\}$$

for some $\tau_0 \in \mathbb{R}$. W.l.o.g., presume $\langle s \to \cdots \to u \rangle \subseteq G(p)$ implying that $u \to_f v_0$ is not in $G(p)$. So, utilizing that the third statement is already proven, we argue

$$\min \underline{F}[u] \leq f_{\langle s \to \cdots \to u \rangle}(\tau_0) \leq f_{\langle s \to \cdots \to u \to_f v_0 \rangle}(\tau_0) < Q.min() \ ,$$

which tells us that u has been removed from Q at least once (due to Lemma 4.18). This means $u \to_f v_0$ has already been relaxed. So, either $u \to_f v_0$ lies in $G(p)$, or $G(p)$ contains an edge $w \to_g v_0$ with $\overline{g} * \overline{F}[w](\tau_0) \leq \underline{f} * \underline{F}[u](\tau_0)$ (see Line 11 and 12 of Algorithm 4.4). The first cannot be the case. So, again utilizing the already proven third statement as well as Equation (3.13) and (3.14), we argue

$$\begin{aligned} \overline{F}[v_0](\tau_0) &\leq \overline{g} * \overline{F}[w](\tau_0) \leq \underline{f} * \underline{F}[u](\tau_0) \leq f * \underline{F}[u](\tau_0) \\ &\leq f * f_{\langle s \to \cdots \to u \rangle}(\tau_0) = f_{\langle s \to \cdots \to u \to_f v_0 \rangle}(\tau_0) < \overline{F}[v_0](\tau_0) \ , \end{aligned}$$

which is a contraction. □

Corollary 4.20. *While approximate TTP search runs,* $\overline{F}[v](\tau) \geq \mathrm{TT}_G(s,v,\tau)$ *is always true for all* $v \in V$ *and all* $\tau \in \mathbb{R}$.

The following simple proof of correctness is quite similar to the correctness proof of time-dependent Dijkstra (see Lemma 4.3 in Section 4.2.1).

Lemma 4.21. *After approximate TTP search terminates,* $\underline{F}[v](\tau) \leq \mathrm{TT}_G(s,v,\tau) \leq \overline{F}[v](\tau)$ *holds for all* $\tau \in \mathbb{R}$ *and all* $v \in V$.

Proof. Assume, there is a node v such that $\underline{F}[v](\tau_0) > \mathrm{TT}_G(s,v,\tau_0)$ for some $\tau_0 \in \mathbb{R}$. Consider a prefix-optimal (s,v,τ_0)-EA-path $\langle s \to \cdots \to v\rangle$ and choose a minimum hop prefix path $\langle s \to \cdots \to u \to_f w\rangle$ fulfilling $\underline{F}[w](\tau_0) > \mathrm{TT}_G(s,w,\tau_0)$, which implies $\underline{F}[u](\tau_0) \leq \mathrm{TT}_G(s,u,\tau_0)$. After relaxing the edge $u \to_f w$ we have

$$\underline{F}[w](\tau_0) \leq f * \underline{F}[u](\tau_0) \leq f * \mathrm{TT}_G(s,u,\tau_0) = \mathrm{TT}_G(s,w,\tau_0) < \underline{F}[w](\tau_0) ,$$

which cannot be the case. The rest follows from Corollary 4.20. □

We not only want to prove that approximate TTP search computes correct lower and upper bounds but also correct predecessor information. This follows directly from Lemma 4.21 together with the second statement of Lemma 4.19.

Corollary 4.22. *After approximate TTP search terminates, the predecessor graph $G(p)$ contains a prefix-optimal (s,v,τ)-EA-path for all $\tau \in \mathbb{R}$ and all reachable nodes $v \in V$.*

As in case of time-dependent Dijkstra, the predecessor information of a node gets permanent as soon as $Q.min()$ gets large enough. This relies on the monotony of $Q.min()$ (see Lemma 4.18).

Lemma 4.23. *The predecessor information $p[v]$ of v does not change anymore as soon as $Q.min() \geq \max\overline{F}[v]$ is fulfilled. From that moment on $G(p)$ is guaranteed to contain a prefix-optimal (s,v,τ)-EA-path for all $\tau \in \mathbb{R}$.*

Proof. As $Q.min()$ never decreases and $\max\overline{F}[v]$ never increases for any node $v \in V$, we can be sure that the condition $Q.min() \geq \max\overline{F}[v]$, once fulfilled, stays true for a node v. Let q_v and m_v be the values that $Q.min()$ and $\max\overline{F}[v]$ have, respectively, at the moment $Q.min() \geq \max\overline{F}[v]$ gets true (i.e., $q_v \geq m_v$). Changing the predecessor information $p[v]$ requires that an edge $u \to_f v$ is relaxed. But this can no more change $p[v]$, since

$$\underline{f} * \underline{F}[u](\tau) \geq \underline{F}[u](\tau) \geq q_v \geq m_v \geq \overline{F}[v](\tau)$$

for all $\tau \in \mathbb{R}$ implies that the condition in Line 11 of Algorithm 4.4 is fulfilled. That the final predecessor information is correct, follows from Corollary 4.22. □

Not only the predecessor information of a node v gets permanent when $Q.min()$ reaches $\max\overline{F}[v]$. The same holds true for the node label.

Lemma 4.24. *The label $\big(\underline{F}[v],\overline{F}[v]\big)$ of v does not change anymore as soon as $Q.min() \geq \max\overline{F}[v]$ is fulfilled. So, $\underline{F}[v](\tau) \leq \mathrm{TT}_G(s,v,\tau) \leq \overline{F}[v](\tau)$ is guaranteed for all $\tau \in \mathbb{R}$ afterwards.*

Proof. If the condition in Line 11 is fulfilled, then relaxing $u \to_f v$ does not change $p[v]$, as we argue in the proof of Lemma 4.23. This also implies that $\big(\underline{F}[v],\overline{F}[v]\big)$ does not change, too. That the final node labels are correct, follows from Lemma 4.21. □

4.2.5 Earliest Arrival Interval Search

Earliest arrival interval search (or *EA interval search* for short, see Algorithm 4.5) is an approximate version of time-dependent Dijkstra (see Section 4.2.1); just like TTP interval search (see Section 4.2.3) is an approximate version of TTP search (see Section 4.2.2). For all $u \in V$ it computes intervals containing $\mathrm{EA}_G(s,u,\tau)$ with $\tau \in [\tau_0, \tau_0']$ for given $s \in V$, $[\tau_0, \tau_0'] \subseteq \mathbb{R}$. The label of a node u is a tentative arrival interval $[q[u], r[u]]$ containing the arrival times of the approximately best paths from s to u found so far. These paths are all contained in the current $G(p)$. When relaxing an edge $u \to_f v$ with lower bound $\underline{f}$ and upper bound $\overline{f}$, we update the label of v by

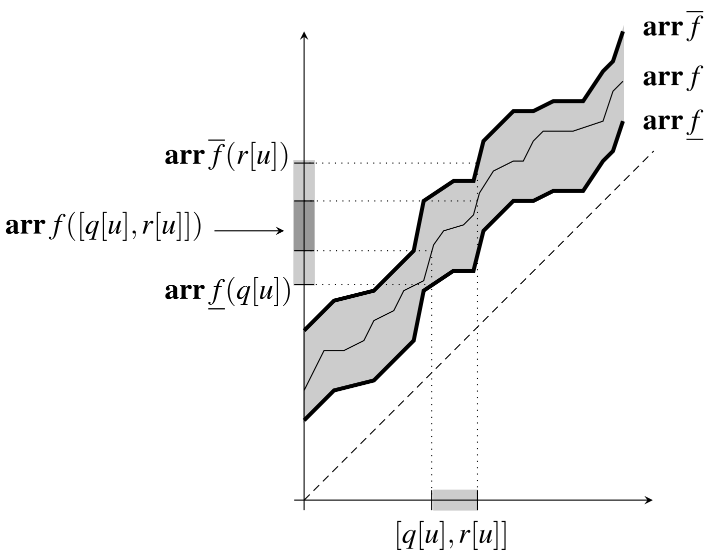
$$[q[v], r[v]] := \Big[\min\big\{q[v], \mathbf{arr}\,\underline{f}(q[u])\big\}, \min\big\{r[v], \mathbf{arr}\,\overline{f}(r[u])\big\}\Big] \,. \tag{4.10}$$

An illustration of $[q_{\mathrm{new}}, r_{\mathrm{new}}] = [\mathbf{arr}\,\underline{f}(q[u]), \mathbf{arr}\,\overline{f}(r[u])]$ is depicted in Figure 4.3. As PQ key of u we use $q[u]$. After termination, the predecessor graph contains a prefix-optimal (s,u,τ)-EA-path for all reachable nodes u and all $\tau \in [\tau_0, \tau_0']$.

The purpose of EA interval search is not to save running time. In fact, we expect a running time similar the running time of time-dependent Dijkstra (see Section 4.2.1). Instead, EA interval search is needed to provide as accurate results

Algorithm 4.5. *Earliest arrival (EA) interval search* is an approximate version of time-dependent Dijkstra (see Algorithm 4.1). For a given start node s and a given departure interval $[\tau_0, \tau_0']$ this algorithm computes labels $[q[u], r[u]] \ni \mathrm{EA}_G(s,u,\tau)$ for all reachable nodes $u \in V$ and all $\tau \in [\tau_0, \tau_0']$.

```
procedure eaIntervalSearch($s : V$, $[\tau_0, \tau_0'] : Interval$)
    $[q[u], r[u]] := [\infty, \infty]$, $p[u] := \emptyset$ for all $u \in V$
    $[q[s], r[s]] := [\tau_0, \tau_0']$
    $Q := \{(s, \tau_0)\} : PriorityQueue$
    while $Q \neq \emptyset$ do
        $u := Q.deleteMin()$
        for $u \to_f v \in E$ with the lower/upper bound $\underline{f}, \overline{f} \in \mathscr{F}_\Pi$ do
            $[q_{\mathrm{new}}, r_{\mathrm{new}}] := \big[\mathbf{arr}\,\underline{f}(q[u]), \mathbf{arr}\,\overline{f}(r[u])\big]$
            if $q_{\mathrm{new}} > r[v]$ then continue
            if $r_{\mathrm{new}} < q[v]$ then $p[v] := \emptyset$          // remove suboptimal predecessors
            $p[v] := \{u\} \cup p[v]$                    // remember a tentative predecessor of v
            if $q_{\mathrm{new}} \geq q[v]$ and $r_{\mathrm{new}} \geq r[v]$ then continue
            $[q[v], r[v]] := \big[\min\{q[v], q_{\mathrm{new}}\}, \min\{r[v], r_{\mathrm{new}}\}\big]$      // update label of v
            if $v \notin Q$ then $Q.insert(v, q[v])$
            else $Q.updateKey(v, q[v])$        // lower bound of the interval is PQ key
```



$\mathbf{arr}\,\overline{f}$
$\mathbf{arr}\,f$
$\mathbf{arr}\,\underline{f}$
$\mathbf{arr}\,\overline{f}(r[u])$
$\mathbf{arr}\,f([q[u],r[u]])$
$\mathbf{arr}\,\underline{f}(q[u])$
$[q[u],r[u]]$

Figure 4.3. How EA interval search obtains $[q_{\mathrm{new}}, r_{\mathrm{new}}] = [\mathbf{arr}\,\underline{f}(q[u]), \mathbf{arr}\,\overline{f}(r[u])]$ from $[q[u], r[u]]$ when an edge $u \to_f v$ is relaxed. If only a lower bound $\underline{f}$ and an upper bound $\overline{f}$ (drawn thick) are present instead of the exact TTF f (drawn thin), then $\mathbf{arr}\,f([q[u], r[u]])$ can not be computed and we compute $[\mathbf{arr}\,\underline{f}(q[u]), \mathbf{arr}\,\overline{f}(r[u])] \supseteq \mathbf{arr}\,f([q[u], r[u]])$ instead.

as possible if only lower and upper bounds $\underline{f}$ and $\overline{f}$ are present in a road network instead of the corresponding exact TTFs f with $u \to_f v \in E$. This is the case in the context of ATCHs (see Section 5.4), a variant of TCHs for fast, exact, and space efficient route planning.

Correctness. To show that the node labels $[q[u], r[u]]$ computed by EA interval search really fulfill $q[u] \leq \mathrm{EA}_G(s, u, \tau) \leq r[u]$ and that $G(p)$ really contains a prefix-optimal (s, u, τ)-EA-path for all $u \in V$ and $\tau \in [\tau_0, \tau_0']$, we begin with the monotonous behavior that is typical for Dijkstra-like algorithms.

Lemma 4.25. *Let $u_1, u_2, \ldots$ be the order in that the nodes are removed from the PQ during EA interval search (nodes may be removed multiple times). Let $[q_1, r_1], [q_2, r_2], \ldots$ be the respective node labels at the time when each node is removed. Then, $q_1 \leq q_2 \leq \ldots$ holds.*

Proof. One can argue by induction over the number of removals from the PQ, analogous to Lemma 4.5. □

Also like for the other Dijkstra-like algorithms, tentative node labels and tentative predecessor information correspond to each other, as the following Lemma shows.

Lemma 4.26. *While EA interval search runs, $G(p)$ contains*

1. *a path $P_{v,\tau} = \langle s \to \cdots \to v\rangle$ with $q[v] \le \mathbf{arr}\, f_{P_{v,\tau}}(\tau) \le r[v]$ for all reached $v \in V$ and all $\tau \in [\tau_0, \tau_0']$,*
2. *all such paths $P_{v,\tau}$ where all prefixes $\langle s \to \cdots \to u\rangle \subseteq P_{v,\tau}$ (including $P_{v,\tau}$ itself) additionally fulfill $\mathbf{arr}\, f_{\langle s \to \cdots \to u\rangle}(\tau) < Q.min()$, and*
3. *no path $R = \langle s \to \cdots \to v\rangle$ with $\mathbf{arr}\, f_R(\tau) < q[v]$ for any $\tau \in [\tau_0, \tau_0']$.*

Proof. The proof follows pretty much the same pattern as the proof of the analogous Lemma in the context of approximate TTP search (see Lemma 4.19). The obvious differences is that labels are intervals instead of pairs of TTFs. Also note that the condition in Line 9 of Algorithm 4.5 is a little stronger than in case of approximate TTP search (see Line 11 of Algorithm 4.4); namely, "$>$" instead of "$\ge$". But this is a very technical detail. □

Corollary 4.27. *While EA interval search runs, $r[v] \ge \mathrm{EA}_G(s, v, \tau)$ is always true for all $v \in V$ and all $\tau \in [\tau_0, \tau_0']$.*

During computation, the node labels $[q[v], r[v]]$ approach $\mathrm{EA}_G(s, v, \tau)$ from above until $\mathrm{EA}_G(s, v, \tau) \in [q[v], r[v]]$ holds in the end for each $v \in V, \tau \in [\tau_0, \tau_0']$.

Lemma 4.28. *After EA interval search terminates, $q[v] \le \mathrm{EA}_G(s, v, \tau) \le r[v]$ holds for all $v \in V$ and all $\tau \in [\tau_0, \tau_0']$.*

Proof. The argument is analogous to Lemma 4.21, but with Corollary 4.27 used in the end. □

Not only the node labels are correct after execution, but the predecessor graph contains an EA path for all reachable nodes and all relevant departure times. This follows directly from Lemma 4.28 together with the second statement of Lemma 4.26.

Corollary 4.29. *After EA interval search terminates, $G(p)$ contains a prefix-optimal (s, v, τ)-EA-path for all $\tau \in [\tau_0, \tau_0']$ and all reachable nodes $v \in V$.*

Again, we are not only interested that node labels and predecessor information are correct after termination of the algorithm, but in the time when the label of a node and the predecessor information get final. This enables us to stop the execution as early as possible. Like for the other Dijkstra-like algorithms, the corresponding necessary condition relies on the monotony of *$Q.min()$* (see Lemma 4.25).

Lemma 4.30. *The label $[q[v], r[v]]$ of v does not change anymore as soon as $Q.min() \ge r[v]$ is fulfilled. So, $q[v] \le \mathrm{EA}_G(s, v, \tau) \le r[v]$ is guaranteed for all $\tau \in [\tau_0, \tau_0']$ afterwards.*

Proof. We argue analogously to the proof Lemma 4.23, although that lemma deals with the predecessor information getting permanent rather than the node labels. The idea behind the argument is, nevertheless, the same because the label and the predecessor information of a node nearly get permanent together. □

The predecessor information $p[v]$ of a node v gets permanent as soon as the slightly stronger condition condition $Q.min() > r[v]$ gets true.

Lemma 4.31. *The predecessor information $p[v]$ of v does not change anymore as soon as $Q.min() > r[v]$ is fulfilled. From that moment on $G(p)$ is guaranteed to contain a prefix-optimal (s,v,τ)-EA-path for all $\tau \in [\tau_0, \tau_0']$.*

Proof. Again, we argue analogously to the proof of Lemma 4.23. Note that the condition $Q.min() > r[v]$ is slightly stronger than the corresponding one in Lemma 4.23, because the condition in Line 9 is a little stronger than the corresponding condition in approximate TTP search. □

Running Time. The running time of EA interval search is similar to the running time of time-dependent Dijkstra (see Section 4.2.1). Reinserts of nodes into the PQ are possible, but they should happen rarely in practice. Their impact on the running time should be negligible hence. The running time of edge relaxation is similar to one of edge relaxation as performed by time-dependent Dijkstra. We simply evaluate two TTFs instead of one (see Line 8 of Algorithm 4.5). Using the bucket structure described in Section 3.1.4, we expect a running time near $\mathrm{O}(|V| \cdot \log |V|)$ in practice.

4.3 Backward Single-Label Searches

This section describes basic time-dependent single label search algorithms that run in backward direction. For constant travel costs, backward search is simple. One simply applies Dijkstra's algorithm to the transpose road network $G^\top$ instead of the original road network G and uses the destination node t as start node. For time-dependent travel costs, things are somewhat more complicated. In most cases, we cannot just apply the forward searches from Section 4.2 to the transpose road network $G^\top$ to obtain suitable algorithms for backward search.

Three of the time-dependent forward single label searches described in Section 4.2—namely, TTP search, TTP interval search, and approximate TTP search—have directly corresponding backward versions; namely, *backward TTP search* (see Section 4.3.1), *backward TTP interval search* (see Section 4.3.2), and *backward approximate TTP search* (see Section 4.3.3). They can be obtained from their forward counterpart by relatively simple modifications. There is also *latest* *departure (LD) interval search* (see Section 4.3.4), a backward search that is dual to EA interval search. It computes the latest possible departure times to reach the destination node t just within a given arrival interval; the corresponding EA paths are also computed of course.

There are also three backward search algorithms that deal with TTFs and additional costs, namely *backward cost profile (CP) search* (see Section 4.3.5), *backward CP interval search* (see Section 4.3.6), and *backward approximate CP search* (see Section 4.3.7). All of them lack a corresponding forward counterpart because of the inconvenient structure of time-dependent travel costs beyond travel times; namely, that the existence of prefix-optimal MC paths is not guaranteed, even in the quite restricted case of time-dependent travel times with additional time-invariant cost (see Section 3.2.1). It is important to note that these three backward search algorithms can not only deal with time-invariant additional costs, but also with more general time-dependent additional costs that are piecewise constant functions (see Section 3.2.4). This is necessary, because backward CP search and backward CP interval search are used in the context of heuristic TCHs where such generalized additional costs can occur (see Section 6.2.2)

Note that all backward searches solve all-to-one problems. This is dual to forward searches, which solve one-to-all problems. The optimal paths to the destination node t computed by a backward search are all contained in the predecessor graphs, just like in case of forward searches. However, a backward search runs on $G^\top$. So, its predecessor graph is not a subgraph of G but of $G^\top$; namely, $G^\top(p) \subseteq G^\top$ with p being the predecessor information. The optimal routes computed by backward searches are thus contained in the *transpose predecessor graph* $(G^\top(p))^\top \subseteq G$.

4.3.1 Backward Travel Time Profile Search

Backward travel time profile search (or *backward TTP search* for short) is the backward version of TTP search [69] (see Algorithm 4.2 in Section 4.2.2). Given a destination node $t \in V$, it runs on $G^\top$ starting from t computing $\mathrm{TT}_G(u,t,\cdot)$ for all $u \in V$. This is dual to forward TTP search, which computes $\mathrm{TT}_G(s,u,\cdot)$ for a given start node s. The node label $F[u]$ is a tentative TTP for traveling from u to t. The transpose tentative predecessor graph $(G^\top(p))^\top \subseteq G$ contains the corresponding tentative EA paths from u to t found so far. When relaxing an edge $u \to_f v \in E^\top$, we update the label $F[v]$ of v by computing

$$F[v] := \min(F[v], F[u] * f) \;. \tag{4.11}$$

The PQ key of a node u is $\min F[u]$. After termination, $(G^\top(p))^\top$ contains a (u,t,τ)-EA-path for all nodes $u \in V$ and all departure times $\tau \in \mathbb{R}$, provided that t is reachable from u in G.

There are only two differences between backward TTP search and its forward counterpart. First, backward TTP search runs on $G^\top$ instead of G. Second, backward TTP search calculates $F[u] * f$ instead of $f * F[u]$ on edge relaxation (compare Equation (4.6) with (4.11)). That f and $F[u]$ must be interchanged, is because linking of TTFs is not commutative in general. Also, backward searches compared to forward searches do no longer consider paths $\langle s \to \cdots \to u \to_f v\rangle$, but paths $\langle v \to_f u \to \cdots \to t\rangle$ in G. So, $F[u]$ comes "after" f on paths from v to t. We do not report pseudocode for backward TTP search, because it is the same as in case of the forward version with small obvious adaptions.

Correctness. Backward TTP search can be regarded as special case of backward approximate CP search, which computes final node labels $\big(\underline{C}[v], \overline{C}[v]\big)$ with $\underline{C}[v](\tau) \leq \mathrm{Cost}_G(v,t,\tau) \leq \overline{C}[v](\tau)$ for all $v \in V$ and all $\tau \in \mathbb{R}$ (see Section 4.3.7). If all additional costs in G are zero, we have $\mathrm{TT}_G(s,t,\cdot) = \mathrm{Cost}_G(s,t,\cdot)$ for all $s,t \in V$. Then, every node label $F[v]$ with respect to backward TTP search corresponds to a node label $\big(\underline{C}[v], \overline{C}[v]\big)$ with respect to backward approximate CP search. This means $F[v](\tau) = \underline{C}[v](\tau) = \mathrm{TT}_G(v,t,\tau) = \overline{C}[v](\tau)$ for all $v \in V$ and all $\tau \in \mathbb{R}$. So, the following statements are essentially special cases of statements elaborately described a little later (see Section 4.3.7). So, proofs are omitted.

Lemma 4.32. *Let $v_1, v_2, \dots$ be the order in that the nodes are removed from the PQ during backward TTP search. Let $F_1, F_2, \dots$ be the respective node labels at time of removal. Then,* $\min F_1 \leq \min F_2 \leq \dots$ *holds.*

Lemma 4.33. *As soon as $Q.min() \geq$* $\max F[v]$ *becomes true for a node v, $F[v]$ and $p[v]$ do not change anymore. From that moment on, $F[v] =$* $\mathrm{TT}_G(v,t,\cdot)$ *holds true and $(G^\top(p))^\top$ is guaranteed to contain a (v,t,τ)-EA-path for all $\tau \in$* $\mathbb{R}$ *(provided that t is reachable from v).*

Lemma 4.34. *After backward TTP search terminates, $F[v] =$* $\mathrm{TT}_G(v,t,\cdot)$ *holds for all $v \in V$. Also, the transpose predecessor graph $(G^\top(p))^\top$ contains a (v,t,τ)-EA-path in G for all $\tau \in$* $\mathbb{R}$ *where t is reachable from v.*

Running Time. All considerations made in the context of forward TTP search can be applied analogously to backward TTP search. We expect backward TTP search to be as slow as its forward counterpart hence.

4.3.2 Backward Travel Time Profile Interval Search

Backward travel time profile interval search (or *backward TTP interval search* for short) is the simplest backward search discussed in this section. Given a destination node $t \in V$, it runs on $G^\top$ starting from t and computes $[\underline{\mathrm{TT}}_G(u,t), \overline{\mathrm{TT}}_G(u,t)]$

for all $u \in V$. This is dual to forward TTP interval search (see Algorithm 4.3 in Section 4.2.3), which computes $[\underline{\mathrm{TT}}_G(s,u), \overline{\mathrm{TT}}_G(s,u)]$ for a start node s.

The difference between forward and backward TTP interval search is very little. One can simply use Algorithm 4.3 replacing E by $E^\top$ (see Line 7) and all occurrences of s by t. We do not report an extra pseudocode for backward TTP interval search hence. The running time needed by backward TTP interval is the same as in case of forward TTP interval search of course. After termination, the transpose predecessor graph $(G^\top(p))^\top \subseteq G$ contains a (u,t,τ)-EA-path for all $u \in V, \tau \in \mathbb{R}$; if t is reachable from u in G.

Correctness. We argue that backward TTP interval search is a special case of backward CP interval search (see Section 4.3.6), which is itself a special case of backward approximate CP search (see Section 4.3.7). The difference between backward TTP interval search and backward CP interval search is that all additional costs are zero. This means that travel costs are travel times and MC paths are EA paths. So, the following statements are essentially special cases of statements described later in this thesis (see Section 4.3.6). So, proofs are omitted.

Lemma 4.35. *Let $v_1, v_2, \dots$ be the order in that the nodes are removed from the PQ during backward TTP interval search. Let $[q_1, r_1], [q_2, r_2], \dots$ be the respective node labels at time of removal. Then, $q_1 \le q_2 \le \dots$ holds.*

Lemma 4.36. *As soon as $Q.min() > r[v]$ becomes true for a node v,*

$$\mathrm{TT}_G(v,t,\tau) \in [q[v], r[v]] = \left[\underline{\mathrm{TT}}_G(v,t), \overline{\mathrm{TT}}_G(v,t)\right]$$

is guaranteed for all $\tau \in \mathbb{R}$. Also, $(G^\top(p))^\top$ contains a (v,t,τ)-EA-path for all $\tau \in \mathbb{R}$ ever after.

Lemma 4.37. *After backward TTP interval search terminates, we can be sure that $[q[v], r[v]] = \left[\underline{\mathrm{TT}}_G(v,t), \overline{\mathrm{TT}}_G(v,t)\right]$ holds for all $v \in V$.*

4.3.3 Backward Approximate Travel Time Profile Search

Backward approximate travel time profile search (or *backward approximate TTP search* for short) is the backward version of approximate TTP search (see Algorithm 4.4 in Section 4.2.4). Given a destination node t, it runs on $G^\top$ starting from t and computes a lower and an upper bound of $\mathrm{TT}_G(u,t,\cdot)$ for every node $u \in V$. The label of a node u is a pair of functions $(\underline{F}[u], \overline{F}[u])$, the tentative lower bound $\underline{F}[u]$ and the tentative upper bound $\overline{F}[u]$. The transpose predecessor graph $(G^\top(p))^\top \subseteq G$ contains corresponding tentative EA paths. Relaxing an edge $u \to_f v \in E^\top$ means that the label $(\underline{F}[v], \overline{F}[v])$ of the node v is updated by

$$\left(\underline{F}[v], \overline{F}[v]\right) := \Big(\min\left(\underline{F}[v], \underline{F}[u] * \underline{f}\right), \min\left(\overline{F}[v], \overline{F}[u] * \overline{f}\right)\Big) . \tag{4.12}$$

After termination, $\underline{F}[u](\tau) \leq \mathrm{TT}_G(u,t,\tau) \leq \overline{F}[u](\tau)$ is fulfilled for all $u \in V$ and all $\tau \in \mathbb{R}$. Also, $(G^\top(p))^\top \subseteq G$ contains a (u,t,τ)-EA-path for all $u \in V$ and all $\tau \in \mathbb{R}$ then; provided that t is reachable from the respective node u.

The difference between the update procedure of backward approximate TTP search (see Equation (4.12)) and forward approximate TTP search (see Equation (4.9)) is that $\underline{f} * \underline{F}[u]$ and $\overline{f} * \overline{F}[u]$ are replaced by $\underline{F}[u] * \underline{f}$ and $\overline{F}[u] * \overline{f}$ respectively. This is analogous to the difference between forward and backward TTP search (see Section 4.3.1). Again, the reason is that linking of TTFs is not commutative and that backward searches consider paths $\langle u \to_f v \to \cdots \to t \rangle$ instead of paths $\langle s \to \cdots \to u \to_f v \rangle$. We do not report pseudocode for backward approximate TTP search, because it is the same as in case of the forward version with the adaptions just explained.

Note that the transpose predecessor graph $(G^\top(p))^\top$ computed by backward approximate TTP search is expected to contain more edges than the one computed by exact backward TTP search (see Section 4.3.1) but less edges than the one computed by backward TTP interval search (see Section 4.3.2)—just like in case of the respective forward search algorithms.

Correctness. We argue that backward approximate TTP search is a special case of backward approximate CP search (see Section 4.3.7). The difference between backward approximate TTP search and backward approximate CP search is that all additional costs are zero. This means that travel costs are travel times and MC paths are EA paths. The following statements are essentially special cases of statements elaborately described a little later (see Section 4.3.7). So, proofs are omitted.

Lemma 4.38. *Let $v_1, v_2, \ldots$ be the order in that the nodes are removed from the PQ during backward approximate TTP search. Let $(\underline{F_1}, \overline{F_1}), (\underline{F_2}, \overline{F_2}), \ldots$ be the respective node labels at time of removal. Then,* $\min \underline{F_1} \leq \min \underline{F_2} \leq \ldots$ *holds.*

Lemma 4.39. *As soon as $Q.min() \geq \max \overline{F}[v]$ becomes true for a node v, we know $(\underline{F}[v], \overline{F}[v])$ and $p[v]$ do not change anymore. From that moment on,*

$$\mathrm{TT}_G(v,t,\tau) \in \big[\underline{F}[v](\tau), \overline{F}[v](\tau)\big]$$

holds and $(G^\top(p))^\top$ is guaranteed to contain a (v,t,τ)-EA-path for all $\tau \in \mathbb{R}$ (provided that t is reachable from v).

Lemma 4.40. *After backward approximate TTP search terminates,*

$$\underline{F}[v](\tau) \leq \mathrm{TT}_G(v,t,\tau) \leq \overline{F}[v](\tau)$$

holds for all $v \in V$ and $\tau \in \mathbb{R}$. Also, the transpose predecessor graph $(G^\top(p))^\top$ contains a (v,t,τ)-EA-path in G for all $\tau \in \mathbb{R}$ where t is reachable from v.

Running Time. All considerations made in the context of forward approximate TTP search can be applied analogously to backward approximate TTP search. We expect backward approximate TTP search to be faster than backward TTP search (see Section 4.3.1), but slower than backward TTP interval search (see Section 4.3.2) hence.

4.3.4 Latest Departure Interval Search

Latest departure interval search (or *LD interval search* for short, see Algorithm 4.6) is dual to EA interval search (see Algorithm 4.5 in Section 4.2.5). Given a destination node t and an arrival time interval $[\sigma_0, \sigma_0']$, it runs on $G^\top$ starting from t and computes intervals containing $\mathrm{LD}_G(u,t,[\sigma_0,\sigma_0'])$ for all $u \in V$. Accordingly, the label of a node u is a tentative departure time interval $[q[u], r[u]]$. The transpose tentative predecessor graph $(G^\top(p))^\top \subseteq G$ contains corresponding tentative EA paths from u to t for every reached node u.

Note that late departure times are better than early ones. So, Q is a maximum PQ instead of a minimum PQ in this case (see Section 2.2.2). Accordingly, we do not remove nodes with minimal key from the PQ, but nodes with maximal key

Algorithm 4.6. *Latest departure (LD) interval search* is dual to EA interval search (see Algorithm 4.5). Given destination node $t \in V$ and an arrival interval $[\sigma_0, \sigma_1'] \subseteq \mathbb{R}$, this Dijkstra-like algorithm computes intervals $[q[v], r[v]] \supseteq \mathrm{LD}_G(v,t,[\sigma_0,\sigma_0'])$ for every node $v \in V$. After termination $(G^\top(p))^\top \subseteq G$ contains a (v,t,τ)-EA-path for all $\tau \in \mathrm{LD}_G(v,t,[\sigma_0,\sigma_0'])$ and all nodes v such that t can be reached from v in G.

```
procedure ldIntervalSearch($s : V$, $[\sigma_0, \sigma_0'] : Interval$)
    $[q[u], r[u]] := [-\infty, -\infty]$, $p[u] := \emptyset$ for all $u \in V$
    $[q[s], r[s]] := [\sigma_0, \sigma_0']$
    $Q := \{(t, \sigma_0)\} : PriorityQueue$
    while $Q \neq \emptyset$ do
        $u := Q.deleteMax()$        // maximizing because later departures are better
        for $u \to_f v \in E^\top$ with the lower/upper bound $\underline{f}, \overline{f} \in \mathscr{F}_\Pi$ do
            $[q_{\text{new}}, r_{\text{new}}] := \left[\min \mathbf{dep}\,\overline{f}(q[u]), \max \mathbf{dep}\,\underline{f}(r[u])\right]$
            if $r_{\text{new}} < q[v]$ then continue
            if $q_{\text{new}} > r[v]$ then $p[v] := \emptyset$
            $p[v] := \{u\} \cup p[v]$
            if $q_{\text{new}} \leq q[v]$ and $r_{\text{new}} \leq r[v]$ then continue
            $[q[v], r[v]] := \left[\max\{q[v], q_{\text{new}}\}, \max\{r[v], r_{\text{new}}\}\right]$
            if $v \notin Q$ then $Q.insert(v, q[v])$
            else $Q.updateKey(v, q[v])$        // keys are increased here, not decreased
```

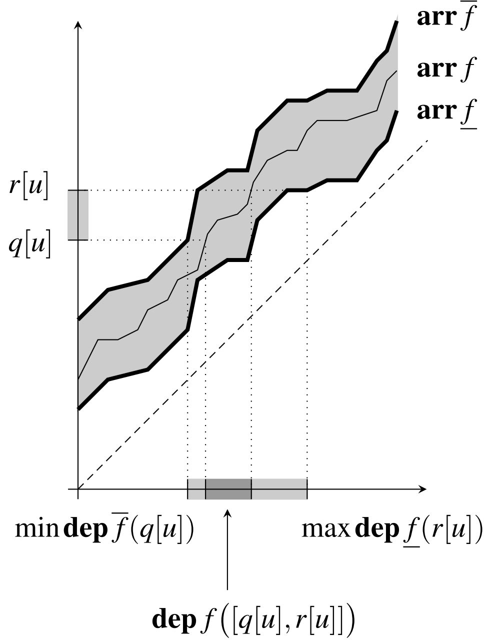


Figure 4.4. How the LD interval search obtains $[q_{\text{new}}, r_{\text{new}}]$ from $[q[u], r[u]]$ when an edge $u \to_f v \in E^\top$ is relaxed. If only a lower bound $\underline{f}$ and an upper bound $\overline{f}$ (drawn thick) are present instead of the exact TTF f (drawn thin), then $\mathbf{dep}\,f\big([q[u], r[u]]\big)$ cannot be computed. Instead we compute $[q_{\text{new}}, r_{\text{new}}] := \big[\min \mathbf{dep}\,\overline{f}(q[u]), \max \mathbf{dep}\,\underline{f}(r[u])\big]$ that contains $\mathbf{dep}\,f\big([q[u], r[u]]\big)$. The interval $[q_{\text{new}}, r_{\text{new}}]$ contains the latest possible times one must depart from v to reach u at times in $[q[u], r[u]]$ if one travels along the edge $v \to_f u \in E$. The use of min and max is necessary because $\mathbf{dep}\,g = (\mathbf{arr}\,g)^{-1}$ is not a one-to-one mapping if a TTF g has one or more segments of slope -1 (i.e., $\mathbf{arr}\,g$ has one or more segments of slope 0).

(see Line 6 where *deleteMax* is invoked instead of *deleteMin*). When relaxing an edge $u \to_f v \in E^\top$ with lower bound $\underline{f}$ and upper bound $\overline{f}$, we update the label and the predecessor information of the node v by

$$[q[v], r[v]] := \Big[\max\big\{q[v], \min \mathbf{dep}\,\overline{f}(q[u])\big\}, \max\big\{r[v], \max \mathbf{dep}\,\underline{f}(r[u])\big\}\Big]\,, \tag{4.13}$$

where $[\min \mathbf{dep}\,\overline{f}(q[u]), \max \mathbf{dep}\,\underline{f}(r[u])]$ is the departure interval at the node v for traveling the corresponding forward edge $v \to_f u \in E$ with arrival interval $[q[u], r[u]]$. The initial label of the node t is $[\sigma_0, \sigma_0']$. After termination, the transpose predecessor graph $(G^\top(p))^\top$ contains a (u, t, τ)-EA-path for all $u \in V$ and all $\tau \in \mathrm{LD}_G(u, t, [\sigma_0, \sigma_0'])$ (provided t is reachable from u). To understand what happens in Equation (4.13), take a look at Figure 4.4. It illustrates how the departure interval $[q_{\text{new}}, r_{\text{new}}] = [\min \mathbf{dep}\,\overline{f}(q[u]), \max \mathbf{dep}\,\underline{f}(r[u])]$ emerges from the arrival interval $[q[u], r[u]]$.

LD interval search is used with approximate TTFs and hence computes approximate results, just like EA interval search. It must be noted, however, that even if all available TTFs are exact and an exact departure time is given, the resulting departure time may still be an interval. The reason is that $\mathbf{dep}\,g$ may not be a one-to-one mapping. This happens if a TTF g has one or more segments with slope -1. However, a slightly different definition of the departure time function of a TTF g,

$$\mathbf{dep}'\,g : \sigma \mapsto \sup \mathbf{dep}\,g(\sigma) ,$$

solves this problem and yields an exact backward version of time-dependent Dijkstra, according to Dean [17].

Correctness. Though LD interval search is dual to EA interval search, their proofs of correctness are not fully analogous. The reason is that LD interval search, as described in this thesis, uses $q[u]$ instead of $r[u]$ as PQ key of a node label $[q[u], r[u]]$. Note that LD interval search solves a maximization problem and not a minimization problem, as it is the case with the other single-label searches. To be fully dual to EA interval search, we should have used $r[u]$ as PQ key hence. This would have enabled us to use $Q.max() < q[u]$ as criterion that the label $[q[u], r[u]]$ of a node u has become final. However, LD interval search is only applied to small cones in this work (see Section 5.4.2) and we do not expect early stopping to bring much benefit hence. Also, the implementation used in the experiments would have required some refactoring, then. So, we opted to keep $q[u]$ as PQ key of a label $[q[u], r[u]]$. For this reason, we omit the discussion whether $Q.max()$ behaves monotonous while LD interval search runs.[1] Instead, we begin with the correspondence of tentative node labels and tentative paths.

Lemma 4.41. *The transpose predecessor graph $(G^\top(p))^\top \subseteq G$ of LD interval search contains,*

1. *while the search runs, for all reached $v \in V$ and all $\sigma \in [\sigma_0, \sigma_0']$, a path $P_{v,\sigma} = \langle v \to \cdots \to t\rangle$ with $\mathbf{dep}\,f_{P_{v,\sigma}}(\sigma) \cap [q[v], r[v]] \neq \emptyset$,*
2. *after termination, all such paths $P_{v,\sigma}$ where all suffixes $\langle u \to \cdots \to t\rangle \subseteq P_{v,\sigma}$ (including $P_{v,\sigma}$ itself) additionally fulfill $\mathbf{dep}\,f_{\langle u \to \cdots \to t\rangle}(\sigma) \cap [q[u], r[u]] \neq \emptyset$, and*
3. *never a path $R = \langle v \to \cdots \to t\rangle$ with $\max \mathbf{dep}\,f_R(\sigma) > r[v]$ for any $\sigma \in [\sigma_0, \sigma_0']$.*

[1] In fact, $Q.max()$ is non-increasing, which is dual to the other single-label searches, where $Q.min()$ is non-decreasing. Using $r[v]$ as PQ key of LD interval search would result in $r_1 \geq r_2 \geq \ldots$ for $[q_i, r_i]$ being the node label on removing the i-th node from the PQ. This would be dual to $q_1 \leq q_2 \leq \ldots$ as in case of EA interval search (see Lemma 4.25).

Proof. That $(G^\top(p))^\top$ contains a path from all reached nodes v to t can be shown analogously to forward searches, where $G(p)$ contains a path from s to every reached node v (see Lemma 4.19).

Assume $v \in V$ and $\sigma \in [\sigma_0, \sigma_0']$ such that $\max \mathbf{dep}\, f_{\langle v \to \cdots \to t\rangle}(\sigma) < q[v]$ holds for all paths $\langle v \to \cdots \to t\rangle \subseteq (G^\top(p))^\top \subseteq G$. There must be an edge $u \to_f v$ in $G^\top(p)$ (i.e., $v \to_f u$ in G) that has been relaxed earlier, such that $q[v] = \min \mathbf{dep}\, \overline{f}(q[u]) \leq \min \mathbf{dep}\, f(q[u])$ holds. W.l.o.g., there is a path $\langle u \to \cdots \to t\rangle \subseteq (G^\top(p))^\top$ that fulfills $\max \mathbf{dep}\, f_{\langle u \to \cdots \to t\rangle}(\sigma) \geq q[u]$. To obtain a contradiction, we argue

$$\begin{aligned}\max \mathbf{dep}\, f_{\langle v \to_f u \to \cdots \to t\rangle}(\sigma) < q[v] \leq \min \mathbf{dep}\, f(q[u]) &\leq \max \mathbf{dep}\, f(q[u]) \\ &\leq \max \mathbf{dep}\, f\big(\max \mathbf{dep}\, f_{\langle u \to \cdots \to t\rangle}(\sigma)\big) \\ &= \max \mathbf{dep}\, \big(f_{\langle u \to \cdots \to t\rangle} * f\big)(\sigma) \\ &= \max \mathbf{dep}\, f_{\langle v \to_f u \to \cdots \to t\rangle}(\sigma)\ .\end{aligned}$$

This works because $\max \mathbf{dep}\, f$ is increasing and because of Equation (3.6).

Next, we show that all paths $P := \langle v \to \cdots \to t\rangle \subseteq (G^\top(p))^\top$ fulfill the condition $\max \mathbf{dep}\, f_P(\sigma) \leq r[v]$ for all $\sigma \in [\sigma_0, \sigma_0']$. Assume a path $P_1 := \langle v \to_f u \to \cdots \to t\rangle \subseteq (G^\top(p))^\top$ with $\max \mathbf{dep}\, f_{P_1}(\sigma_1) > r[v]$ for some $\sigma_1 \in [\sigma_0, \sigma_0']$. W.l.o.g., presume $\max \mathbf{dep}\, f_{\langle u \to \cdots \to t\rangle}(\sigma_1) \leq r[u]$. The edge $v \to_f u$ lies in $(G^\top(p))^\top$, which means $u \to_f v \in E^\top$ has already been relaxed. To obtain a contradiction again, we argue

$$\begin{aligned}\max \mathbf{dep}\, f_{P_1}(\sigma_1) > r[v] \geq \max \mathbf{dep}\, \underline{f}(r[u]) &\geq \max \mathbf{dep}\, f(r[u]) \\ &\geq \max \mathbf{dep}\, f\big(\mathbf{dep}\, f_{\langle u \to \cdots \to s\rangle}(\sigma_1)\big) = \max \mathbf{dep}\, f_{P_1}(\sigma_1)\ .\end{aligned}$$

Now we know that the first and the third statement both hold true. To show the second statement, assume a path $\langle v \to_f u \to \cdots \to t\rangle \not\subseteq (G^\top(p))^\top$ such that all its suffixes $\langle u' \to \cdots \to t\rangle \subseteq \langle v \to_f u \to \cdots \to t\rangle$ fulfill

$$\mathbf{dep}\, f_{\langle u' \to \cdots \to t\rangle}(\sigma) \cap [q[u'], r[u']] \neq \emptyset$$

for some $\sigma \in [\sigma_0, \sigma_0']$. W.l.o.g., presume $\langle u \to \cdots \to t\rangle \subseteq (G^\top(p))^\top$ implying that $v \to_f u \in E$ is not in $(G^\top(p))^\top$. After termination, $u \to_f v \in E^\top$ has been relaxed at least once. So, either $v \to_f u$ lies in $(G^\top(p))^\top$, or $(G^\top(p))^\top$ contains an edge $v \to_g w$, such that $w \to_g v \in E^\top$ has been relaxed earlier, with $\min \mathbf{dep}\, \overline{g}(q[w]) > \max \mathbf{dep}\, \underline{f}(r[u])$ implying

$$\begin{aligned}q[v] \geq \min \mathbf{dep}\, \overline{g}(q[w]) &> \max \mathbf{dep}\, \underline{f}(r[u]) \\ &\geq \max \mathbf{dep}\, f\big(\max \mathbf{dep}\, f_{\langle u \to \cdots \to t\rangle}(\sigma)\big) \\ &= \max \mathbf{dep}\, f_{\langle v \to_f u \to \cdots \to t\rangle}(\sigma) \geq q[v]\ .\end{aligned}$$

In either case this is contrary to the assumptions. □

Note that the second statement of Lemma 4.41 is not fully dual to the second statement of the corresponding Lemma 4.26. Again, this is due to the fact that we use $q[v]$ instead of $r[v]$ as PQ key of a node v. This has consequences for the rest of the correctness proof, which we formulate without an early stopping criterion like $Q.max() < q[v]$. The following two Lemmas can only be applied after termination of LD interval search hence.

Lemma 4.42. *After LD interval search terminates,* $[q[v], r[v]] \supseteq \mathrm{LD}_G(v,t,[\sigma_0,\sigma_0'])$ *holds for all* $v \in V$.

Proof. Assume $v \in V$ exists with $r[v] < \tau_0' := \max \mathrm{LD}_G(v,t,\sigma_0')$ after termination. Consider a (v,t,τ_0')-EA-path $\langle v \rightarrow_f u \rightarrow \cdots \rightarrow t\rangle$. W.l.o.g., presume $r[u] \geq \max \mathrm{LD}_G(u,t,\sigma_0')$ (otherwise, choose an appropriate suffix path of $\langle v \rightarrow_f u \rightarrow \cdots \rightarrow t\rangle$). The edge $u \rightarrow_f v \in E^\top$ must have been relaxed. So,

$$\begin{aligned} r[v] &\geq \max \mathbf{dep}\, \underline{f}(r[u]) \geq \max \mathbf{dep}\, f\big(\max \mathrm{LD}_G(u,t,\sigma_0')\big) \\ &= \max \big((\mathbf{arr}\, f)^{-1} \circ \mathrm{EA}_G^{-1}(u,t,\cdot)\big)(\sigma_0') \\ &= \max \big(\mathrm{EA}_G(u,t,\cdot) \circ \mathbf{arr}\, f\big)^{-1}(\sigma_0') \end{aligned}$$

holds true. Moreover, we have

$$\sigma_0' = \mathrm{EA}_G(v,t,\tau_0') = \mathrm{EA}_G\big(u,t,\mathbf{arr}\, f(\tau_0')\big) = \mathrm{EA}_G(u,t,\cdot) \circ \mathbf{arr}\, f(\tau_0')$$

because of the suffix-optimality of EA paths. Together with

$$\big(\mathrm{EA}_G(u,t,\cdot) \circ \mathbf{arr}\, f\big)^{-1} \circ \mathrm{EA}_G(u,t,\cdot) \circ \mathbf{arr}\, f(\tau_0') \ni \tau_0'$$

this yields $r[v] \geq \tau_0' = \max \mathrm{LD}_G(v,t,\sigma_0')$—a contradiction.

Further, assume $q[v] > \tau_0 := \min \mathrm{LD}_G(v,t,\sigma_0)$. There must be an edge $v \rightarrow_f u$ in G with $q[v] = \min \mathbf{dep}\, \overline{f}(q[u])$ (e.g., the transpose of the edge $u \rightarrow_f v \in E^\top$ that has been relaxed lastly during LD interval search). W.l.o.g., presume $q[u] \leq \min \mathrm{LD}_G(u,t,\sigma_0)$ (otherwise, set $v := u$, consider another edge $v \rightarrow_f u$ in G with $q[v] = \min \mathbf{dep}\, \overline{f}(q[u])$, and repeat this until $q[u] \leq \min \mathrm{LD}_G(u,t,\sigma_0)$ gets true). Now, we argue

$$\begin{aligned} q[v] &\leq \min \mathbf{dep}\, f(q[u]) \leq \min \mathbf{dep}\, f\big(\mathrm{LD}_G(u,t,\sigma_0)\big) \\ &= \min \big(\mathrm{EA}_G(u,t,\cdot) \circ \mathbf{arr}\, f\big)^{-1}(\sigma_0) \end{aligned}$$

and with $\sigma_0 = \mathrm{EA}_G(v,t,\tau_0) \leq \mathrm{EA}_G(u,t,\mathbf{arr}\, f(\tau_0)) = \mathrm{EA}_G(u,t,\cdot) \circ \mathbf{arr}\, f(\tau_0)$ we obtain

$$q[v] \leq \min \big(\mathrm{EA}_G(u,t,\cdot) \circ \mathbf{arr}\, f\big)^{-1} \circ \mathrm{EA}_G(u,t,\cdot) \circ \mathbf{arr}\, f(\tau_0) \leq \tau_0 \;.$$

But this means $q[v] \leq \tau_0 = \min \mathrm{LD}_G(v,t,\sigma_0)$—a contradiction. □

Lemma 4.43. *After LD interval search terminates, $(G^{\top}(p))^{\top} \subseteq G$ contains a (v,t,τ)-EA-path for all $v \in V$ from that t is reachable and all $\tau \in \mathrm{LD}_G(v,t,[\sigma_0,\sigma_0'])$.*

Proof. Consider a (v,t,τ)-EA-path $P_{v,\tau}$ in G, which exists for $v \in V$ and $\tau \in \mathrm{LD}_G(v,t,[\sigma_0,\sigma_0'])$ due to Lemma 3.7. Then, we have

$$\mathbf{arr}\, f_{\langle u \to \cdots \to t\rangle}(\tau_{vu}) = \mathrm{EA}_G(u,t,\tau_{vu}) = \mathrm{EA}_G(v,t,\tau) \in [\sigma_0,\sigma_0']$$

with $\tau_{vu} := \mathbf{arr}\, f_{\langle v \to \cdots \to u\rangle}(\tau)$ for all suffixes $\langle u \to \cdots \to t\rangle \subseteq P_{v,\tau}$, because of the suffix-optimality of EA paths. This implies that $\sigma \in [\sigma_0,\sigma_0']$ exists such that $\tau_{vu} \in \mathbf{dep}\, f_{\langle u \to \cdots \to t\rangle}(\sigma) \cap \mathrm{LD}_G(u,t,\sigma)$ holds for all the suffixes. So, as the LD interval search has already terminated, we have

$$\emptyset \neq \mathbf{dep}\, f_{\langle u \to \cdots \to t\rangle}(\sigma) \cap \mathrm{LD}_G(u,t,\sigma) \subseteq \mathbf{dep}\, f_{\langle u \to \cdots \to t\rangle}(\sigma) \cap [q[u],r[u]]$$

for each suffix, because Lemma 4.42 guaranties $\mathrm{LD}_G(u,t,\sigma) \subseteq [q[u],r[u]]$. But this means $P_{v,\tau} \subseteq (G^{\top}(p))^{\top}$, which follows from the second statement of Lemma 4.41. □

Running Time. LD interval search is dual to EA interval search, which means we expect similar running times. More precisely, we expect that reinserts of nodes happen rarely. This results in about $\mathrm{O}(|V|\log|V|)$ running time plus the time needed for $\mathrm{O}(|E|) = \mathrm{O}(|V|)$ edge relaxations, assuming G to be sparse. Relaxing an edge $u \to_f v \in E^{\top}$ needs constant time except for computing the interval $\big[\min \mathbf{dep}\, \overline{f}(q[u]), \max \mathbf{dep}\, \underline{f}(q[u])\big]$. The latter needs $\mathrm{O}\big(|\overline{f}| + |\underline{f}|\big)$ time if a linear scan through the bend points of $\overline{f}$ and $\underline{f}$ is used. Binary search would reduce this to $\mathrm{O}\big(\log|\overline{f}| + \log|\underline{f}|\big)$ time. But we expect $|\overline{f}|$ and $|\underline{f}|$ to be small. A linear scan through the bend points of $\overline{f}$ and $\underline{f}$ should be fast enough hence—at least in the context of ATCHs (see Section 5.4.1), which is where LD interval search is used in this thesis (see Section 5.4.2).

4.3.5 Backward Cost Profile Search

Backward cost profile search (or *backward CP search* for short, see Algorithm 4.7) is the first Dijkstra-like algorithm in this chapter that not only deals with time-dependent travel times but also with additional costs. So, G has no longer edges $u \to_f v$ but edges $u \to_{f|c} v$ with $f|c \in \mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$. But backward CP search also works with the more general case that additional costs are time-dependent; that is, edge weights $f|c$ are taken from $\mathscr{F}_\Pi^1 \times \mathscr{X}_\Pi$ and c is a function then. This is important because backward CP search is utilized by heuristic TCH preprocessing, where such generalized edge weights may really occur (see Section 6.2.2 on page 280).

Given a destination node t, backward CP search computes $\mathrm{MCTT}_G(u,t,\cdot)$ as well as $\mathrm{Cost}_G(u,t,\cdot) - \mathrm{MCTT}_G(u,t,\cdot)$ and thus implicitly $\mathrm{Cost}_G(u,t,\cdot)$ for all $u \in V$. The search runs backward starting from t. A node u is labeled with a pair $F[u]\big|C[u] \in \mathscr{F}_\Pi^1 \times \mathscr{X}_\Pi$, where $F[u]$ and $C[u]$ represent the travel times and the additional cost, respectively, of the best paths found so far. So, $F[u] + C[u]$ is a tentative CP for traveling from u to t. The transpose tentative predecessor graph $(G^\top(p))^\top \subseteq G$ contains the corresponding tentative paths from u to t. After termination, $F[u] = \mathrm{MCTT}_G(u,t,\cdot)$ and $C[u] = \mathrm{Cost}_G(u,t,\cdot) - \mathrm{MCTT}_G(u,t,\cdot)$ is fulfilled for all $u \in V$, which implies $F[u] + C[u] = \mathrm{Cost}_G(u,t,\cdot)$. Also, $(G^\top(p))^\top$ contains a (u,t,τ)-MC-path for all $\tau \in \mathbb{R}$ if t is reachable from the node u.

Like in case of LD interval search, edges are relaxed in backward direction; that is, relaxed edges $u \to_{f|c} v$ are elements of $E^\top$ instead of E. When relaxing an edge $u \to_{f|c} v \in E^\top$, we update the label $F[v]\big|C[v]$ of the node v by

$$F[v]\big|C[v] := \min\big(F[v]\big|C[v],\ F[u]\big|C[u] \star f|c\big)\ . \tag{4.14}$$

Note that we link $F[u]\big|C[u] \star f|c$ and not $f|c \star F[u]\big|C[u]$, because the search direction is backward (see Line 8). As PQ key of a node u we would use the minimum of the current tentative CP, but this does in general not exist because of

Algorithm 4.7. Given a destination node t, backward CP search computes final labels $F[u]\big|C[u]$ with $F[u] + C[u] = \mathrm{Cost}_G(u,t,\cdot)$ for all $u \in V$. The components of the label represent the travel times and the additional costs of corresponding MC paths; that is, $F[u] = \mathrm{MCTT}_G(u,t,\cdot)$ and $C[u] = \mathrm{MCTT}_G(u,t,\cdot) - \mathrm{Cost}_G(u,t,\cdot)$. After termination $(G^\top(p))^\top \subseteq G$ contains a (u,t,τ)-MC-path for all $\tau \in \mathbb{R}$ (if t is reachable from u).

```
1  procedure bwCpSearch($t : V$)
2      $F[u]|C[u] := \infty|\infty$ for all $u \in V$, $F[t]|C[t] :\equiv 0|0$        // labels from $\mathscr{F}_\Pi^1 \times \mathscr{X}_\Pi$
3      $p[u] := \emptyset$ for all $u \in V$                  // predecessor information consists of sets
4      $Q := \{(t,0)\}$ : PriorityQueue
5      while $Q \neq \emptyset$ do
6          $u := Q.deleteMin()$
7          foreach $u \to_{f|c} v \in E^\top$ do
8              $g_{\mathrm{new}}|d_{\mathrm{new}} := F[u]|C[u] \star f|c$
9              if $g_{\mathrm{new}} + d_{\mathrm{new}}(\tau) > F[v] + C[v](\tau)$ for all $\tau \in \mathbb{R}$ then continue
10             if $g_{\mathrm{new}} + d_{\mathrm{new}}(\tau) < F[v] + C[v](\tau)$ for all $\tau \in \mathbb{R}$ then $p[v] := \emptyset$
11             $p[v] := \{u\} \cup p[v]$
12             $F[v]|C[v] := \min(F[v]|C[v],\ g_{\mathrm{new}}|d_{\mathrm{new}})$
13             if $v \notin Q$ then $Q.insert(v, \inf(F[v] + C[v]))$
14             else $Q.updateKey(v, \inf(F[v] + C[v]))$
```

$F[u]\big|C[u] \in \mathscr{F}_{\Pi}^{1} \times \mathscr{X}_{\Pi}$. Instead, we use $\inf(F[u]+C[u])$ (see Line 13 and 14), which surely exists. Like in case of TTP search, reinserts into the PQ are possible.

Correctness. Though backward CP search can be considered as a special case of backward approximate CP search (see Section 4.3.7), we cannot argue its correctness from that. The reason is that there are, essentially, two ways to formulate exact and approximate variants of backward CP search.

- First, the search computes information about the TTFs and the ACFs as two separate functions. The information about TCFs is then given implicitly by the information about TTFs and ACFs.
- Second, the search only computes the TCFs. Reconstructing the TTFs and the ACFs cannot be done easily then.

Backward CP search as described in this thesis deals with TTFs and ACFs separately. Backward approximate CP search as described in this thesis only deals with TCFs.

The proof of correctness of backward CP search follows the same pattern as in case of the other Dijkstra-like algorithms. But note that the existence of MC paths may be problematic if edge weights are taken from $\mathscr{F}_{\Pi}^{1} \times \mathscr{X}_{\Pi}$. Although we strongly conjecture that the existence of MC paths is guaranteed in this setup, we have not worked out a proof so far (see Section 3.2.4). To ensure that an (s,t,τ_0)-MC-path exists for all $s,t \in V, \tau_0 \in \mathbb{R}$ where t is reachable from s, we use Lemma 3.23 instead. So, in the following, $\inf f > 0$ or $\min c > 0$ is assumed to hold true for all edges $u \rightarrow_{f|c} v$ in G.

Analogous to all Dijkstra-like algorithms considered in this thesis, backward CP search shows the typical monotonous increase of PQ keys.

Lemma 4.44. *Let $v_1, v_2, \ldots$ be the order in that the nodes are removed from the PQ during backward CP search. Let $F_1|C_1, F_2|C_2, \ldots$ be the respective node labels at time of removal. Then,* $\inf(F_1+C_1) \leq \inf(F_2+C_2) \leq \ldots$ *holds.*

Proof. The proof is analogous to the one Lemma 4.54 but with the difference that TCFs are represented implicitly as pairs of TTFs and ACFs. □

This means, $Q.min()$ never decreases during backward CP search. And, obviously, $\sup(F[v]+C[v])$ never increases for any node $v \in V$. This yields the usual criterion when the label and the predecessor information of a node get final, as discussed a little later in this section. Also, we find the usual correspondence between node labels and tentative predecessor information.

Lemma 4.45. *While backward CP search runs, $(G^{\top}(p))^{\top}$ contains a path $P_{v,\tau} = \langle v \rightarrow \cdots \rightarrow t \rangle$ with $f_{P_{v,\tau}}(\tau) = F[v](\tau)$ and $c_{P_{v,\tau}}(\tau) = C[v](\tau)$ for all $v \in V, \tau \in \mathbb{R}$ (provided that t is reachable from v).*

Proof. Assume, there is $v_0 \in V, \tau_0 \in \mathbb{R}$ such that

$$f_{\langle v_0 \to \cdots \to t\rangle}(\tau_0) \neq F[v_0](\tau_0) \quad \text{or} \quad c_{\langle v_0 \to \cdots \to t\rangle}(\tau_0) \neq C[v_0](\tau_0)$$

holds for all paths $\langle v_0 \to \cdots \to t\rangle \subseteq (G^\top(p))^\top \subseteq G$. There must be an earlier relaxed edge $u \to_{f|c} v_0 \in E^\top$ with $v_0 \to_{f|c} u$ in $(G^\top(p))^\top \subseteq G$ that fulfills

$$\begin{aligned} F[v_0](\tau_0) &= \lim_{\sigma \to \tau_0^-} F[u] * f(\sigma) = F[u] * f(\tau_0) \quad \text{and} \\ C[v_0](\tau_0) &= \lim_{\sigma \to \tau_0^-} C[u] *_f c(\sigma) = C[u] *_f c(\tau_0) \end{aligned}$$

with respect to Equation (3.39) and (3.41), because all involved functions are left-continuous. W.l.o.g., there is a path $\langle u \to \cdots \to t\rangle \subseteq (G^\top(p))^\top$ with

$$\begin{aligned} f_{\langle u \to \cdots \to t\rangle}(\mathbf{arr}\, f(\tau_0)) &= F[u](\mathbf{arr}\, f(\tau_0)) \quad \text{and} \\ c_{\langle u \to \cdots \to t\rangle}(\mathbf{arr}\, f(\tau_0)) &= C[u](\mathbf{arr}\, f(\tau_0)) \,. \end{aligned}$$

So, to obtain a contradiction, we argue

$$\begin{aligned} f_{\langle v_0 \to_{f|c} u \to \cdots \to t\rangle}(\tau_0) \neq F[v_0](\tau_0) &= F[u] * f(\tau_0) = F[u](\mathbf{arr}\, f(\tau_0)) + f(\tau_0) \\ &= f_{\langle u \to \cdots \to t\rangle}(\mathbf{arr}\, f(\tau_0)) + f(\tau_0) = f_{\langle v_0 \to_{f|c} u \to \cdots \to t\rangle}(\tau_0) \end{aligned}$$

and analogously $c_{\langle v_0 \to_{f|c} u \to \cdots \to t\rangle}(\tau_0) \neq c_{\langle v_0 \to_{f|c} u \to \cdots \to t\rangle}(\tau_0)$. □

Corollary 4.46. *While backward CP search runs, $F[v] + C[v](\tau) \geq \mathrm{Cost}_G(v,t,\tau)$ is always true for all $v \in V, \tau \in \mathbb{R}$.*

We are now ready to show that backward CP search computes the desired node labels.

Lemma 4.47. *After backward CP search terminates, $F[v] + C[v] = \mathrm{Cost}_G(v,t,\cdot)$ and $F[v] = \mathrm{MCTT}_G(v,t,\cdot)$ holds for all $v \in V$.*

Proof. We start our proof with the assumption that $v_1 \in V, \tau_1 \in \mathbb{R}$ exist such that $F[v_1] + C[v_1](\tau_1) \neq \mathrm{Cost}_G(v_1,t,\tau_1)$ holds. Consider a (v_1,t,τ_1)-MC-path

$$P_1 := \langle v_1 \to_{f_1|c_1} v_2 \to_{f_2|c_2} \cdots \to_{f_{k-1}|c_{k-1}} v_k = t\rangle \subseteq G \,.$$

We write $P_i := \langle v_i \to_{f_i|c_i} \cdots \to_{f_{k-1}|c_{k-1}} v_k = t\rangle \subseteq P_1$ to denote the suffixes of P_1 for all $i \in \{2,\ldots,k\}$. We also write $\tau_i := \mathbf{arr}\, f_{i-1}(\tau_{i-1})$. Due to the suffix-optimality of MC paths, we know P_i is an (v_i,t,τ_i)-MC-path for all $i \in \{2,\ldots,k\}$. Choose j maximal such that

$$F[v_j] + C[v_j](\tau_j) \neq \mathrm{Cost}_G(v_j,t,\tau_j)$$

but

$$F[v_{j+1}] + C[v_{j+1}](\tau_{j+1}) = \mathrm{Cost}_G(v_{j+1}, t, \tau_{j+1})$$

holds. Corollary 4.46 tells us that $F[v_j] + C[v_j](\tau_j) > \mathrm{Cost}_G(v_j, t, \tau_j)$ holds. Let F_{old} and C_{old} be the values of $F[v_j]$ and $C[v_j]$, respectively, right before the transpose edge $v_{j+1} \to_{f_j|c_j} v_j \in E^\top$ is relaxed for the last time. To obtain a contradiction, we argue

$$F[v_j] + C[v_j](\tau) \le \mathbf{tcf}\min\big(F_{\mathrm{old}}\big|C_{\mathrm{old}},\, F[v_{j+1}]\big|C[v_{j+1}] \star f_j|c_j\big)(\tau)$$

for all $\tau \in \mathbb{R}$ implying

$$\begin{aligned}
\mathrm{Cost}_G(v_j, t, \tau_j) &< F[v_j] + C[v_j](\tau_j) \\
&\le \min\big\{F_{\mathrm{old}} + C_{\mathrm{old}}(\tau_j),\, \big(F[v_{j+1}] * f_j + C[v_{j+1}] *_{f_j} c_j\big)(\tau_j)\big\} \\
&\le F[v_{j+1}] * f_j + C[v_{j+1}] *_{f_j} c_j(\tau_j) \\
&= (F[v_{j+1}] + C[v_{j+1}]) \circ \mathbf{arr}\, f_j + f_j + c_j(\tau_j) \\
&= \mathrm{Cost}_G(v_{j+1}, t, \tau_{j+1}) + f_j(\tau_j) + c_j(\tau_j) \\
&= \mathrm{Cost}_G(v_j, t, \tau_j)\ .
\end{aligned}$$

Now, we assume that $F[v_1](\tau_1) \ne \mathrm{MCTT}_G(v_1, t, \tau_1)$ holds, which is a little more complicated to deal with. Again, consider the path P_1 and its suffixes $P_2, \ldots, P_k$, but this time assuming that $\delta > 0$ exists such that each path P_i is a (v_i, t, τ)-MC-path for all $\tau \in (\tau_i - \delta, \tau_i]$ with $i \in \{1, \ldots, k\}$. This is possible because of Lemma 3.25 and the suffix-optimality of MC paths. Further assume that each path P_i fulfills $f_{P_i}(\tau_i) = \mathrm{MCTT}_G(v_i, t, \tau_i)$. Choose j maximal, such that $F[v_j](\tau_j) \ne \mathrm{MCTT}_G(v_j, t, \tau_j)$ but $F[v_{j+1}](\tau_{j+1}) = \mathrm{MCTT}_G(v_{j+1}, t, \tau_{j+1})$ holds. Note that we already know $F[v_j] + C[v_j] = \mathrm{Cost}_G(v_j, t, \cdot)$ implying $F[v_j](\tau_j) > \mathrm{MCTT}(v_j, t, \tau_j)$ because of Lemma 4.45 and Equation (3.29).

Let, again, $F_{\mathrm{old}}\big|C_{\mathrm{old}}$ be the value of $F[v_j]\big|C[v_j]$ right before $v_{j+1} \to_{f_j|c_j} v_j \in E^\top$ is relaxed for the last time. We then calculate

$$\begin{aligned}
&\big(F[v_{j+1}] * f_j + C[v_{j+1}] *_{f_j} c_j\big)(\tau) \\
&\qquad = F[v_{j+1}] + C[v_{j+1}](\mathbf{arr}\, f_j(\tau)) + f_j(\tau) + c_j(\tau) \\
&\qquad = \mathrm{Cost}_G(v_{j+1}, t, \mathbf{arr}\, f_j(\tau)) + f_j(\tau) + c_j(\tau) \\
&\qquad = C_{P_{j+1}}(\mathbf{arr}\, f_j(\tau)) + f_j(\tau) + c_j(\tau) \\
&\qquad = C_{P_j}(\tau) \\
&\qquad = \mathrm{Cost}(v_j, t, \tau) \\
&\qquad \le F_{\mathrm{old}} + C_{\mathrm{old}}(\tau)
\end{aligned}$$

for all $\tau \in (\tau_j - \delta, \tau_j)$ and $\mathbf{arr}\, f_j(\tau) \in (\mathbf{arr}\, f_j(\tau_j - \delta), \tau_{j+1})$ with sufficiently small $\delta > 0$. So, it is enough to distinguish whether $\big(F[v_{j+1}] * f_j + C[v_{j+1}] *_{f_j} c_j\big)(\tau)$ is smaller or equal $F_{\text{old}} + C_{\text{old}}(\tau)$ for all $\tau \in (\tau_j - \delta, \tau_j)$, again with sufficiently small $\delta > 0$. In the first case, we obtain

$$\begin{aligned} F[v_j](\tau_j) &= \min_{C_{\text{old}}, C[v_{j+1}] *_{f_j} c_j}(F_{\text{old}}, F[v_{j+1}] * f_j)(\tau_j) \\ &= \lim_{\tau \to \tau_j^-} F[v_{j+1}] * f_j(\tau) = F[v_{j+1}] * f_j(\tau_j) \ . \end{aligned}$$

In the second case, we obtain

$$F[v_j](\tau_j) = \lim_{\tau \to \tau_j^-} \min\big(F_{\text{old}}, F[v_{j+1}] * f_j\big)(\tau) \leq F[v_{j+1}] * f_j(\tau_j) \ .$$

Altogether, this yields

$$\begin{aligned} \text{MCTT}_G(v_j, t, \tau_j) &= f_{P_j}(\tau_j) = f_{P_{j+1}} * f_j(\tau_j) = f_{P_{j+1}}(\tau_{j+1}) + f_j(\tau_j) \\ &= \text{MCTT}_G(v_{j+1}, t, \tau_{j+1}) + f_j(\tau_j) = F[v_{j+1}](\tau_{j+1}) + f_j(\tau_j) \\ &= F[v_{j+1}] * f_j(\tau_j) \geq F[v_j](\tau_j) \\ &> \text{MCTT}_G(v_j, t, \tau_j) \ , \end{aligned}$$

which is a contradiction. □

Lemma 4.45 together with Lemma 4.47 finally provides the desired result.

Lemma 4.48. *As soon as $Q.min() > \sup(F[v] + C[v])$ becomes true for a node v, $F[v] \big| C[v]$ and $p[v]$ do not change anymore. From that moment on,*

$$F[v] = \text{MCTT}_G(v, t, \cdot) \quad \textit{and} \quad C[v] = \text{Cost}_G(v, t, \cdot) - \text{MCTT}_G(v, t, \cdot)$$

holds and $(G^\top(p))^\top$ contains a (v, t, τ)-MC-path for all $\tau \in \mathbb{R}$ ever after.

Proof. The argument is analogous to the proofs of Lemma 4.23 and 4.24. □

We summarize some parts of what we have learned in a corollary.

Corollary 4.49. *After backward CP search terminates, $F[v] = \text{MCTT}_G(v, t, \cdot)$ and $C[v] = \text{Cost}_G(v, t, \cdot) - \text{MCTT}_G(v, t, \cdot)$ holds for all $v \in V$. Also, the transpose predecessor graph $(G^\top(p))^\top$ contains a (v, t, τ)-MC-path in G for all $\tau \in \mathbb{R}$ if t is reachable from v.*

Running Time. As backward CP search is similar to TTP search, one would expect similar running times. However, CPs can get much more complex than TTPs. More precisely, if K is the total number of bend points in G, then TTPs have no more than $K \cdot |V|^{O(\log |V|)}$ bend points according to Foschini et al. [36]. CPs, in contrast, have up to $2^{\Omega(|V|)}$ bend points, as we state in Theorem 6.2 (see Section 6.1.2). So, backward CP search can take even more time than the already slow TTP search, because even more bend points can emerge that have to be processed. Whether backward CP search is slower than TTP search for road networks in practice, must be found out experimentally.

One-to-One Version. Like other Dijkstra-like algorithms, backward CP search can be used to answer one-to-one queries. To compute $\mathrm{Cost}_G(s,t,\cdot)$ for given $s,t \in V$ (i.e., to answer a CP query), one simply runs backward CP search starting from t returning the TCF $F[s]+C[s]$ after termination. Of course, it is not necessary to run backward CP search completely, but it can be stopped as soon as $Q.min() > \sup(F[s]+C[s])$ becomes true, because the label $F[s]\big|C[s]$ of the node s is guaranteed to be final from that time on.

Computing TCFs Only. We already said that Section 4.3.7 describes a version of backward approximate CP search that only deals with TCFs instead of TTFs and ACFs. It would also be possible to formulate backward CP search this way.

4.3.6 Backward Cost Profile Interval Search

As backward CP search (see Algorithm 4.7 in Section 4.3.5) is very slow, we use *backward cost profile interval search* (or *backward CP interval search* for short, see Algorithm 4.8) as a relatively loose but much faster approximation of backward CP search; just like TTP interval search is a relatively loose but fast approximation of TTP search.

Algorithm 4.8. Given a destination node t, backward CP interval search computes final labels $[q[u], r[u]] = [\underline{\mathrm{Cost}}_G(u,t), \overline{\mathrm{Cost}}_G(u,t)]$ for all $\tau \in \mathbb{R}$ and all $u \in \mathbb{R}$. After termination, $(G^\top(p))^\top \subseteq G$ contains an (u,t,τ)-MC-path for each node $u \in V$ from that t is reachable in G and all $\tau \in \mathbb{R}$.

```
procedure $bwCpIntervalSearch(t : V)$
    $[q[u], r[u]] := [\infty, \infty]$ for all $u \in V$, $[q[t], r[t]] := [0,0]$        // labels are intervals
    $p[u] := \emptyset$ for all $u \in V$                     // predecessor information consists of sets
    $Q := \{(t,0)\} : PriorityQueue$
    while $Q \neq \emptyset$ do
        $u := Q.deleteMin()$
        foreach $u \to_{f|c} v \in E^\top$ do
            $[q_{\text{new}}, r_{\text{new}}] := \big[q[u] + \inf(f+c), r[u] + \sup(f+c)\big]$
            if $q_{\text{new}} > r[v]$ then continue
            if $r_{\text{new}} < q[v]$ then $p[v] := \emptyset$          // remove suboptimal predecessors
            $p[v] := \{u\} \cup p[v]$                  // remember a tentative predecessor of v
            if $q_{\text{new}} \geq q[v]$ and $r_{\text{new}} \geq r[v]$ then continue
            $[q[v], r[v]] := \big[\min\{q[v], q_{\text{new}}\}, \min\{r[v], r_{\text{new}}\}\big]$     // update label of v
            if $v \notin Q$ then $Q.insert(v, q[v])$
            else $Q.updateKey(v, q[v])$        // lower bound of the interval is PQ key
```

Instead of $\mathrm{MCTT}_G(u,t,\cdot)$ and $\mathrm{Cost}_G(u,t,\cdot) - \mathrm{MCTT}_G(s,u,\cdot)$ it computes the interval $[\underline{\mathrm{Cost}}_G(u,t), \overline{\mathrm{Cost}}_G(u,t)]$ for every node $u \in V$. Hence, the label of a node u is neither a tentative CP nor a pair taken from $\mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$ but a tentative interval $[q[u], r[u]]$. The transpose tentative predecessor graph $(G^\top(p))^\top \subseteq G$ contains corresponding tentative MC paths from all reached nodes to the destination node t for all possible departure times. When relaxing an edge $u \to_f v \in E^\top$ we update the label of a node v by

$$\begin{aligned}[q[v], r[v]] := \Big[&\min\big\{q[v],\, q[u] + \inf(f+c)\big\}, \\ &\min\big\{r[v],\, r[u] + \sup(f+c)\big\}\Big]\,.\end{aligned} \tag{4.15}$$

Note that, compared to the analogous Equation (4.8), "min" and "max" are replaced by "inf" and "sup" respectively. This is because of $f|c \in \mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$, which means that $\min(f+c)$ and $\max(f+c)$ do in general not exist. As PQ key of a node u we use $q[u]$. Like in case of backward CP search, nodes may also be reinserted into the PQ from time to time. But we expect this to happen rarely in practice.

Correctness. We argue that backward CP interval search is a special case of backward approximate CP search (see Section 4.3.7) where the lower and the upper bound $\underline{C}[u]$ and $\overline{C}[u]$ of a node u, respectively, are simply constant functions. Most of the following statements are essentially special cases of statements elaborately described a little later (see Section 4.3.7). So, proofs are omitted.

Lemma 4.50. *Let $v_1, v_2, \ldots$ be the order in that the nodes are removed from the PQ during backward CP interval search. Let $[q_1, r_1], [q_2, r_2], \ldots$ be the respective node labels at time of removal. Then, $q_1 \le q_2 \le \ldots$ holds.*

Lemma 4.51. *As soon as $Q.min() > r[v]$ becomes true for a node v, $[q[v], r[v]]$ and $p[v]$ do not change anymore and $\mathrm{Cost}_G(v,t,\tau) \in [q[v], r[v]]$ is guaranteed for all $\tau \in \mathbb{R}$ from that moment on. Also, $(G^\top(p))^\top$ contains a (v,t,τ)-MC-path for all $\tau \in \mathbb{R}$ ever after.*

The final label $[q[v], r[v]]$ of a node v is not only some interval containing $\mathrm{Cost}_G(v,t,\tau)$ for all $\tau \in V$. In fact, it is the smallest possible interval that can be computed in terms of minima and maxima of TCFs. This is analogous to forward and backward TTP interval search.

Lemma 4.52. *After backward CP interval search terminates, we know $[q[v], r[v]] = \big[\underline{\mathrm{Cost}}_G(v,t), \overline{\mathrm{Cost}}_G(v,t)\big]$ holds for all $v \in V$.*

Proof. The proof is analogous to the one of Lemma 4.15. □

Lemma 4.52 enables us to formulate the statement of Lemma 4.51 in a more specific way, just like in case of forward and backward TTP interval search.

Corollary 4.53. *As soon as $Q.min() > r[v]$ becomes true for a node v,*

$$\mathrm{Cost}_G(v,t,\tau) \in [q[v], r[v]] = \left[\underline{\mathrm{Cost}}_G(v,t), \overline{\mathrm{Cost}}_G(v,t)\right]$$

is guaranteed for all $\tau \in \mathbb{R}$. Also, $(G^\top(p))^\top$ contains a (v,t,τ)-MC-path for all $\tau \in \mathbb{R}$ ever after.

Running Time. Backward CP interval search is very similar to TTP interval search. Reinserts into the PQ are possible, but like in case of TTP interval search we do not expect that this happens too often in practice. We expect similar running times as in case of Dijkstra's algorithm hence.

One-to-One Version. Of course, backward CP interval search can be used to answer one-to-one queries. To compute the interval $\left[\underline{\mathrm{Cost}}_G(s,t), \overline{\mathrm{Cost}}_G(s,t)\right]$ for given $s,t \in V$, one only has to run backward CP interval search starting from t returning $[q[s], r[s]]$ after termination. The search can be stopped as soon as $Q.min() > r[s]$ is fulfilled.

4.3.7 Backward Approximate Cost Profile Search

Backward approximate cost profile search (or *backward approximate CP search* for short, see Algorithm 4.9) is, like backward CP interval search, an approximate version of backward CP search. Backward approximate CP search is far more accurate than backward CP interval search but runs significantly slower. However, it is still expected to run much faster than backward CP search. The relationship of backward CP search, backward CP interval search, and backward approximate CP search is analogous to the relationship of TTP search, TTP interval search, and approximate TTP search. The idea behind backward approximate CP search is to use approximate TTFs and ACFs as well as approximate node labels instead of exact ones. Expecting the approximate functions to have smaller complexity than the exact ones, we expect the search to run faster, because much less bend points have to be processed. This idea is analogous to the idea behind approximate TTP search of course. Note that backward approximate CP search not only works with additional time-invariant costs but also with time-dependent additional costs; that is, $f|c \in \mathscr{F}_\Pi^1 \times \mathscr{X}_\Pi$ can hold for edges $u \to_{f|c} v \in E$. Also note that $f + c$ may even have points of discontinuity. This can happen in the context of heuristic TCHs even if the underlying road networks has edge weights only taken from $\mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$ (see Section 6.2.2 and 6.3.2).

Algorithm 4.9. Given a destination node t, approximate backward CP search computes final labels $\left(\underline{C}[u], \overline{C}[u]\right)$ with $\underline{C}[u](\tau) \leq \mathrm{Cost}_G(u,t,\tau) \leq \overline{C}[u](\tau)$ for all $u \in V$. After termination, $(G^\top(p))^\top \subseteq G$ contains a (u,t,τ)-MC-path for all $\tau \in \mathbb{R}$ and each node u from that the destination node t is reachable in G.

```
**procedure** *bwApproximateCpSearch*$(t : V)$
    $\underline{C}[u] := \overline{C}[u] := \infty$ for all $u \in V$
    $\underline{C}[s] := \overline{C}[s] :\equiv 0|0$                    *// labels from $\mathscr{F}_\Pi \times \mathscr{F}_\Pi$*
    $p[u] := \emptyset$ for all $u \in V$          *// predecessor information consists of sets*
    $Q := \{(t,0)\}$ : *PriorityQueue*
    **while** $Q \neq \emptyset$ **do**
        $u := Q.deleteMin()$
        **foreach** $u \rightarrow_{f|c} v \in E^\top$ **do**
            let $\overline{f} \in \mathscr{F}_\Pi^{\downharpoonright}$ be an upper bound of $f$ with $\left|\overline{f}\right| \ll |f|$
            let $\overline{c} \in \mathscr{X}_\Pi$ be an upper bound of $c$ with $|\overline{c}| \ll |c|$
            let $\underline{f} \in \mathscr{F}_\Pi^{\downharpoonright}$ be a lower bound of $f$ with $\left|\underline{f}\right| \ll |f|$
            let $\underline{c} \in \mathscr{X}_\Pi$ be a lower bound of $c$ with $|\underline{c}| \ll |c|$
            $\left(\underline{D}_{\text{new}}, \overline{D}_{\text{new}}\right) := \left(\underline{C}[u] *_{\underline{f}} \left(\underline{f} + \underline{c}\right), \overline{C}[u] *_{\overline{f}} \left(\overline{f} + \overline{c}\right)\right)$
            **if** $\underline{D}_{\text{new}}(\tau) \geq \overline{C}[v](\tau)$ for all $\tau \in \mathbb{R}$ **then continue**
            **if** $\overline{D}_{\text{new}}(\tau) < \underline{C}[v](\tau)$ for all $\tau \in \mathbb{R}$ **then** $p[v] := \emptyset$
            $p[v] := \{u\} \cup p[v]$
            **if** $\underline{D}_{\text{new}}(\tau) \geq \underline{C}[v](\tau) \wedge \overline{D}_{\text{new}}(\tau) \geq \overline{C}[v](\tau)$ for all $\tau \in \mathbb{R}$ **then continue**
            $\left(\underline{C}[v], \overline{C}[v]\right) := \left(\min\left(\underline{C}[v], \underline{D}_{\text{new}}\right), \min\left(\overline{C}[v], \overline{D}_{\text{new}}\right)\right)$
            **if** $\left|\overline{C}[v]\right|$ gets too large **then**
                replace $\overline{C}[v]$ by an an upper bound from $\mathscr{F}_\Pi$ with less complexity
            **if** $\left|\underline{C}[v]\right|$ gets too large **then**
                replace $\underline{C}[v]$ by an an upper bound from $\mathscr{F}_\Pi$ with less complexity
            **if** $\overline{C}[v]$ has one or more discontinuities **then**
                replace $\overline{C}[v]$ by an upper bound $\overline{C}' \in \mathscr{F}_\Pi$ with $\max \overline{C}' = \sup \overline{C}[v]$
            **if** $\underline{C}[v]$ has one or more discontinuities **then**
                replace $\underline{C}[v]$ by a lower bound $\underline{C}' \in \mathscr{F}_\Pi$ with $\min \underline{C}' = \inf \underline{C}[v]$
            **if** $v \notin Q$ **then** $Q.insert(v, \min \underline{C}[v])$
            **else** $Q.updateKey(v, \min \underline{C}[v])$
```

For all nodes $u \in V$ from that t is reachable, backward approximate CP search computes node labels of the form $\big(\underline{C}[u], \overline{C}[u]\big) \in \mathscr{F}_\Pi \times \mathscr{F}_\Pi$ that fulfill

$$\underline{C}[u](\tau) \leq \mathrm{Cost}_G(u,t,\tau) \leq \overline{C}[u](\tau)$$

for all $\tau \in \mathbb{R}$ after termination. Also, the transpose predecessor graph $(G^\top(p))^\top$ contains a (u,t,τ)-MC-path for all $\tau \in \mathbb{R}$ and all nodes u from that t is reachable in the end. As PQ key of a node u we use $\min \underline{C}[u]$. Reinserts into the PQ are possible. When an edge $u \rightarrow_{f|c} v \in E^\top$ is relaxed (see Line 18), the label of a node v is updated by

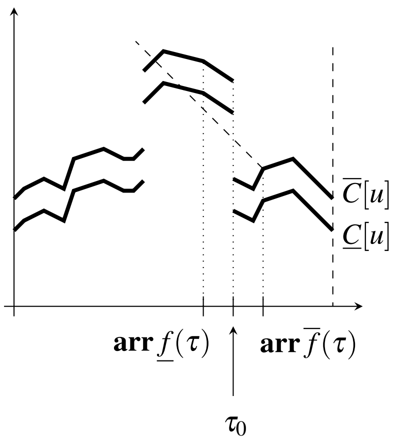
$$\begin{aligned}\big(\underline{C}[v], \overline{C}[v]\big) := \Big(&\min\Big(\underline{C}[v],\, \underline{C}[u] *_{\underline{f}} \big(\underline{f} + \underline{c}\big)\Big), \\ &\min\Big(\overline{C}[v],\, \overline{C}[u] *_{\overline{f}} \big(\overline{f} + \overline{c}\big)\Big)\Big)\end{aligned} \tag{4.16}$$

where $\underline{f}$ and $\overline{f}$ as well as $\underline{c}$ and $\overline{c}$ are lower and upper bounds of f as well as c, respectively; that is, $\underline{f}(\tau) \leq f(\tau) \leq \overline{f}(\tau)$ and $\underline{c}(\tau) \leq c(\tau) \leq \overline{c}(\tau)$ for all $\tau \in \mathbb{R}$. The bounds $\underline{f}, \overline{f}, \underline{c}, \overline{c}$ (see Line 9 to 12) can either be computed during the execution of the algorithm or in a preprocessing step, just like in case of approximate TTP search. This already makes $\underline{C}[u]$ and $\overline{C}[u]$ lower and upper bounds for every reached node u respectively. The running time can, again, be further reduced if $\underline{C}[u]$ and $\overline{C}[u]$ are further approximated during execution (see Lines 19 to 22).

After Line 26, $\underline{C}[u]$ and $\overline{C}[u]$ are guaranteed to lie in $\mathscr{F}_\Pi$, which means they formally fulfill the FIFO property. Otherwise, the linking in Line 13 could yield TCFs $\underline{D}_{\text{new}}$ and $\overline{D}_{\text{new}}$ that fail to be a lower and upper bound, respectively, of the CP D_{new} that emerges from the best paths $\langle u \rightarrow \cdots \rightarrow t \rangle \subseteq G$ considered so far. As a simple example, assume that an edge $u \rightarrow_{f|0} v \in E^\top$ is relaxed, and $\big(\underline{D}_{\text{new}}, \overline{D}_{\text{new}}\big)$ is thus computed by

$$\big(\underline{D}_{\text{new}}, \overline{D}_{\text{new}}\big) := \Big(\underline{C}[u] *_{\underline{f}} \big(\underline{f} + 0\big), \overline{C}[u] *_{\overline{f}} \big(\overline{f} + 0\big)\Big) = \big(\underline{C}[u] * \underline{f}, \overline{C}[u] * \overline{f}\big)\ .$$

Further assume, $\underline{C}[u]$ and $\overline{C}[u]$ have a negative discontinuity—such that $\underline{C}[u]$ and $\overline{C}[u]$ violate the FIFO property—for some departure time $\tau_0 \in \mathbb{R}$ where the width of the downward jump is "sufficiently large". Then, it can happen that $\underline{D}_{\text{new}}$ lies not completely below $\overline{D}_{\text{new}}$. Especially, $\underline{D}_{\text{new}}$ or $\overline{D}_{\text{new}}$ cannot be a lower or upper bound of D_{new}, respectively, because the condition $\underline{D}_{\text{new}}(\tau) \leq D_{\text{new}}(\tau) \leq \overline{D}_{\text{new}}(\tau)$ is violated for the one or another $\tau \in \mathbb{R}$. Figure 4.5 illustrates such a situation. There, $\mathbf{arr}\,\underline{f}(\tau)$ and $\mathbf{arr}\,\overline{f}(\tau)$ are to the left and to the right of the discontinuity τ_0



$\overline{C}[u]$
$\underline{C}[u]$
$\mathbf{arr}\,\underline{f}(\tau)$
$\mathbf{arr}\,\overline{f}(\tau)$
τ_0

Figure 4.5. How negative discontinuities destroy the property of being a lower or an upper bound of a TCF when linked. The negative discontinuity that the lower bound $\underline{C}[u]$ and the upper bound $\overline{C}[u]$ of an unknown TCF have at τ_0, implies that the FIFO property is violated. To fulfill $\big(\underline{C}[u] * \underline{f}\big)(\tau) \leq \big(\overline{C}[u] * \overline{f}\big)(\tau)$ for a lower bound $\underline{f}$ and an upper bound $\overline{f}$ of an unknown TTF and for some departure time τ, the value $\underline{C}[u](\mathbf{arr}\,\underline{f}(\tau))$ must not lie above the dashed line of slope -1. This is because $\underline{C}[u](\mathbf{arr}\,\underline{f}(\tau)) > \overline{C}[u](\mathbf{arr}\,\overline{f}(\tau)) + \overline{f}(\tau) - \underline{f}(\tau)$ is fulfilled there.

respectively. Consider the equivalence

$$\begin{array}{rrcl} & \big(\underline{C}[u] * \underline{f}\big)(\tau) & \leq & \big(\overline{C}[u] * \overline{f}\big)(\tau) \\ \Leftrightarrow & \big(\underline{C}[u] \circ \mathbf{arr}\,\underline{f} + \underline{f}\big)(\tau) & \leq & \big(\overline{C}[u] \circ \mathbf{arr}\,\overline{f} + \overline{f}\big)(\tau) \\ \Leftrightarrow & \underline{C}[u]\big(\mathbf{arr}\,\underline{f}(\tau)\big) & \leq & \overline{C}[u]\big(\mathbf{arr}\,\overline{f}(\tau)\big) + \overline{f}(\tau) - \underline{f}(\tau)\ . \end{array}$$

This shows that $\underline{C}[u] * \underline{f}(\tau) \leq \overline{C}[u] * \overline{f}(\tau)$ is not fulfilled because of

$$\underline{C}[u]\big(\mathbf{arr}\,\underline{f}(\tau)\big) > \overline{C}[u]\big(\mathbf{arr}\,\overline{f}(\tau)\big) + \overline{f}(\tau) - \underline{f}(\tau)\ .$$

To avoid such problems, $\underline{C}[u]$ and $\overline{C}[u]$ are replaced by bounds taken from $\mathscr{F}_\Pi$, which fulfill the FIFO property (see Line 23 to 26).

The computation of the bounds $\overline{f}$, $\overline{c}$, $\underline{f}$, and $\underline{c}$ (see Line 9 to 12) can be done using the Imai-Iri algorithm [52], just like in case of approximate TTP search (see Section 4.2.4). This algorithm can also be used when $\underline{C}[u]$ or $\overline{C}[u]$, respectively, are replaced by other lower and upper bounds, which happens if $\big|\underline{C}[u]\big|$ or $\big|\overline{C}[u]\big|$ get too large (see Line 19 to 22) or if $\underline{C}[u]$ or $\overline{C}[u]$ have points of discontinuity as just explained (see Line 23 to 26).

Computing Travel Time and Additional Cost Separately. There is an important difference between backward approximate CP search as described here and

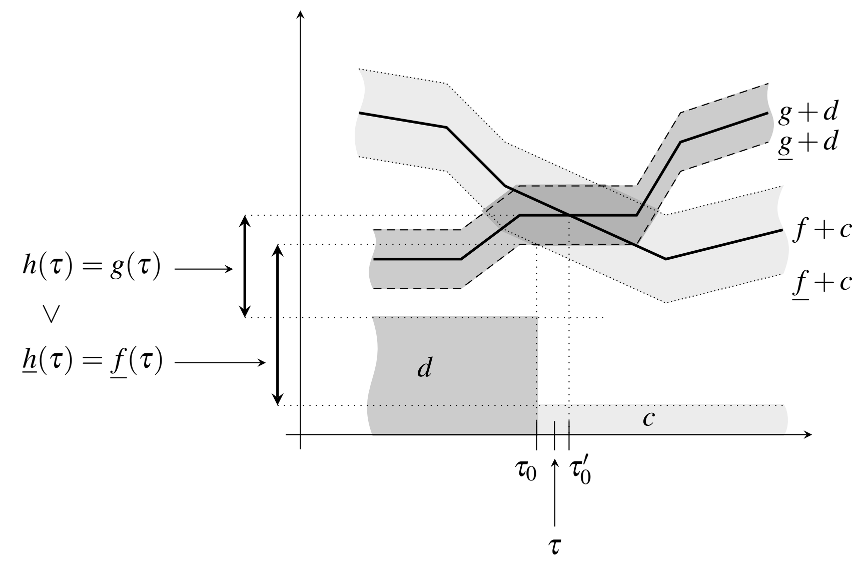


Figure 4.6. How the minimum operation on $\mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$ destroys the property of being a lower bound of a TTF. Consider $f|c, g|d \in \mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$ and corresponding lower bounds $\underline{f}, \underline{g} \in \mathscr{F}_\Pi$ of the TTFs f, g. The lower bounds $\underline{f}+c$ and $\underline{g}+d$ of the TCFs $f+c$ and $g+d$ intersect at τ_0, which is different from τ_0' where $f+c$ and $g+d$ intersect. This means, that the points of discontinuity of $h|e := \min\big(f|c,\, g|d\big)$ are different from the ones of $\underline{h}|\underline{e} := \min\big(\underline{f}|c,\, \underline{g}|d\big)$; h is discontinuous at τ_0', and $\underline{h}$ at $\tau_0 \neq \tau_0'$. For all $\tau \in (\tau_0, \tau_0')$, $\underline{h}$ fails to be a lower bound of h, because of $\underline{h}(\tau) = \underline{f}(\tau) > g(\tau) = h(\tau)$.

backward CP search as described in Section 4.3.5. Namely, that backward CP search as described there does not directly compute labels $C[u] = \operatorname{Cost}(u,t,\cdot)$, but labels $F[u]\big|C[u] = \operatorname{MCTT}(u,t,\cdot)\big|\operatorname{Cost}(u,t,\cdot) - \operatorname{MCTT}(u,t,\cdot)$. It would also be possible to formulate backward approximate CP search as direct generalization of this approach by computing labels of the form

$$\Big(\underline{F}[u]\big|\underline{C}[u],\, \overline{F}[u]\big|\overline{C}[u]\Big) \in \big(\mathscr{F}^1_\Pi \times \mathscr{X}_\Pi\big)^2$$

that fulfill

$$\big(\underline{F}[u] + \underline{C}[u]\big)(\tau) \leq \operatorname{Cost}_G(u,t,\tau) \leq \big(\overline{F}[u] + \overline{C}[u]\big)(\tau)$$

and

$$\underline{F}[u](\tau) \leq \operatorname{MCTT}_G(u,t,\tau) \leq \overline{F}[u](\tau)$$

for all $\tau \in \mathbb{R}$ after termination. This, however, is not completely straightforward but requires some care.

Consider, for example, $f|c,\, g|d \in \mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$ as well as corresponding lower bounds $\underline{f}, \underline{g} \in \mathscr{F}_\Pi$ of the TTFs f, g. In general, the lower bound $\underline{h}$, with $\underline{h}|\underline{e} :=$

$\min\big(\underline{f}|c, \underline{g}|d\big)$, is not a lower bound of h, with $h|e := \min\big(f|c, g|d\big)$. This is because points of discontinuity shift in general. Figure 4.6 illustrates such a situation. So, one cannot naively apply the minimum operation on $\mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$ to obtain appropriate lower bounds but has to be more careful. For upper bounds of TTFs as well as for lower and upper bounds of ACFs this problem exists analogously.

Correctness. Proving the correctness of backward approximate CP search is more or less analogous to proving the correctness of approximate TTP search (see Section 4.2.4). We start with the monotonous behavior that is typical for all other Dijkstra-like algorithms considered in this thesis.

Lemma 4.54. *Let $v_1, v_2, \dots$ be the order in that the nodes are removed from the PQ during backward approximate CP search (nodes may be removed multiple times). Let $\big(\underline{C_1}, \overline{C_1}\big), \big(\underline{C_2}, \overline{C_2}\big), \dots$ be the respective node labels at the time when each node is removed. Then,* $\min \underline{C_1} \leq \min \underline{C_2} \leq \dots$ *holds.*

Proof. Analogous to the proof of Lemma 4.5, we argue by induction over the number i of removals from the PQ. In the base case (i.e., $i = 2$) we remove u_2 from the PQ which has been inserted by relaxing an edge $t \rightarrow_{f|c} u_2$. We have

$$\min \underline{C_2} = \inf\Big(\underline{C_1} *_{\underline{f}} \big(\underline{f} + \underline{c}\big)\Big) = \inf\big(0 \circ \mathbf{arr}\,\underline{f} + \underline{f} + \underline{c}\big) \geq 0 = \min \underline{C_1}\ .$$

Now we assume

$$\min \underline{C_1} \leq \min \underline{C_2} \leq \cdots \leq \min \underline{C_{i-1}}$$

and consider what happens when u_i is removed. The label of node u_i has been updated lastly by relaxing an edge $u_j \rightarrow_{g|d} u_i \in E^\top$ with $j < i$. So, we have

$$\min \underline{C_i} = \inf\Big(\underline{C_j} *_{\underline{g}} \big(\underline{g} + \underline{d}\big)\Big) = \inf\Big(\underline{C_j} \circ \mathbf{arr}\,\underline{g} + \underline{g} + \underline{d}\Big) \geq \min \underline{C_j}\ .$$

For $j = i - 1$, this means $\min \underline{C_{i-1}} \leq \min \underline{C_i}$. Otherwise, u_i is already contained in the PQ when u_{i-1} is removed, which implies $\min \underline{C_{i-1}} \leq \min \underline{C_i}$. $\square$

As usual, $Q.min()$ never decreases during the execution of backward approximate CP search. Together with the fact that $\max \overline{C}[v]$ never increases for any node v, this yields the typical criterion to determine when node labels and predecessor information get permanent.

Of course, the above proof follows pretty much the same pattern as the proof of Lemma 4.5. But together with Section 4.3.5, this section is the only part in this chapter closely considering the correctness of a Dijkstra-like backward search that deals with time-dependent travel costs beyond travel times. We opted to present the above proof more detailed hence. The same applies for the proof of Lemma 4.55 below.

Lemma 4.55. *While backward approximate CP search runs,* $(G^\top(p))^\top$ *contains*

1. *a path* $P_{v,\tau} = \langle v \to \cdots \to t\rangle$ *with* $\underline{C}[v](\tau) \le C_{P_{v,\tau}}(\tau) \le \overline{C}[v](\tau)$ *for all reached nodes* v *and all* $\tau \in \mathbb{R}$,
2. *all such paths* $P_{v,\tau}$ *where all suffixes* $\langle u \to \cdots \to t\rangle \subseteq P_{v,\tau}$ *(including* $P_{v,\tau}$ *itself) additionally fulfill* $C_{\langle u\to\cdots\to t\rangle}(\tau_u) < \min\{\overline{C}[u](\tau_u), Q.min()\}$ *with* $\tau_u := \mathbf{arr}\, f_{\langle v\to\cdots\to u\rangle}(\tau)$, *and*
3. *no path* $R = \langle v \to \cdots \to t\rangle$ *with* $C_R(\tau) < \underline{C}[v](\tau)$ *for any* $v \in V, \tau \in \mathbb{R}$.

Proof. That $(G^\top(p))^\top$ contains a path from all reached nodes v to t follows analogously to forward searches (see the proof of Lemma 4.19).

Assume there is $v_0 \in V$ and $\tau_0 \in \mathbb{R}$ such that $C_{\langle v_0\to\cdots\to t\rangle}(\tau_0) > \overline{C}[v_0](\tau_0)$ holds for all paths $\langle v_0 \to \cdots \to t\rangle \subseteq (G^\top(p))^\top$. There must be an earlier relaxed edge $u \to_{f|c} v_0$ in $G^\top(p)$ with $\overline{C}[v_0](\tau_0) = \overline{C}[u] *_{\overline{f}} (\overline{f} + \overline{c})(\tau_0)$. W.l.o.g., there is a path $\langle u \to \cdots \to t\rangle \subseteq (G^\top(p))^\top$ fulfilling $C_{\langle u\to\cdots\to t\rangle}(\mathbf{arr}\, f(\tau_0)) \le \overline{C}[u](\mathbf{arr}\, f(\tau_0))$. So, to obtain a contradiction, we argue

$$\begin{aligned} C_{\langle v_0 \to_{f|c} u\to\cdots\to t\rangle}(\tau_0) &> \overline{C}[v_0](\tau_0) = \overline{C}[u] *_{\overline{f}} (\overline{f} + \overline{c})(\tau_0) \\ &\ge C_{\langle u\to\cdots\to t\rangle} *_f (f + c)(\tau_0) = C_{\langle v_0 \to_{f|c} u\to\cdots\to t\rangle}(\tau_0) \end{aligned}$$

utilizing that $\overline{C}[u]$ fulfills the FIFO property (this is enforced by Line 23 to 26 of Algorithm 4.9).

Next, we show that all paths $P := \langle v \to \cdots \to t\rangle \subseteq (G^\top(p))^\top$ fulfill $C_P(\tau) \ge \underline{C}[v](\tau)$ for all $\tau \in \mathbb{R}$. So, assume $P_0 := \langle v \to_{f|c} u \to \cdots \to t\rangle \subseteq (G^\top(p))^\top$ with $C_{P_0}(\tau_0) < \underline{C}[v_0](\tau_0)$ for some $\tau_0 \in \mathbb{R}$. W.l.o.g., presume $C_{\langle u\to\cdots\to t\rangle}(\mathbf{arr}\, f(\tau_0)) \ge \underline{C}[u](\mathbf{arr}\, f(\tau_0))$. The edge $v_0 \to_f u$ lies in $(G^\top(p))^\top$, which means its transpose version has already been relaxed. To obtain a contradiction, we utilize $\underline{C}[u] \in \mathscr{F}_\Pi$ and argue

$$\begin{aligned} C_{P_0}(\tau_0) < \underline{C}[v_0](\tau_0) &\le \underline{C}[u] *_{\underline{f}} (\underline{f} + \underline{c})(\tau_0) \\ &\le C_{\langle u\to\cdots\to t\rangle} *_f (f + c)(\tau_0) = C_{P_0}(\tau_0) \ . \end{aligned}$$

At this point, we know that the first and the third statement must both be true. To show the second statement, assume a path $\langle v_0 \to_{f|c} u \to \cdots \to t\rangle \not\subseteq (G^\top(p))^\top$ such that all its suffixes $\langle u' \to \cdots \to t\rangle \subseteq \langle v_0 \to_{f|c} u \to \cdots \to t\rangle$ (including the path itself) fulfill the condition

$$C_{\langle u'\to\cdots\to t\rangle}\big(\mathbf{arr}\, f_{\langle v_0\to\cdots\to u'\rangle}(\tau_0)\big) < \min\big\{\overline{C}[u']\big(\mathbf{arr}\, f_{\langle v_0\to\cdots\to u'\rangle}(\tau_0)\big), Q.min()\big\}$$

for some $\tau_0 \in \mathbb{R}$. W.l.o.g., presume $\langle u \to \cdots \to t\rangle \subseteq (G^\top(p))^\top$ implying that $v_0 \to_{f|c} u$ is not in $(G^\top(p))^\top$. So, utilizing that the third statement is already proven, we argue

$$\min \underline{C}[u] \le C_{\langle u\to\cdots\to t\rangle}(\mathbf{arr}\, f(\tau_0)) \le C_{\langle v_0 \to_{f|c} u\to\cdots\to t\rangle}(\tau_0) < Q.min() \ ,$$

which tells us that u has been removed from Q at least once (due to Lemma 4.54). This means $u \rightarrow_{f|c} v_0 \in E^\top$ has already been relaxed. So, either $v_0 \rightarrow_{f|c} u$ lies in $(G^\top(p))^\top$, or $(G^\top(p))^\top$ contains an edge $v_0 \rightarrow_{g|d} w$ with $\overline{C}[w] *_{\overline{g}} \left(\overline{g}+\overline{d}\right)(\tau_0) \leq \underline{C}[u] *_{\underline{f}} \left(\underline{f}+\underline{c}\right)(\tau_0)$ (see Line 14 and 15 of Algorithm 4.9). The first cannot be the case. So, utilizing the already proven third statement, we argue

$$\begin{aligned}\overline{C}[v_0](\tau_0) &\leq \overline{C}[w] *_{\overline{g}} \left(\overline{g}+\overline{d}\right)(\tau_0) \leq \underline{C}[u] *_{\underline{f}} \left(\underline{f}+\underline{c}\right)(\tau_0)\\ &\leq C_{\langle u \rightarrow \cdots \rightarrow t\rangle} *_f (f+c)(\tau_0) = C_{\langle v_0 \rightarrow_{f|c} u \rightarrow \cdots \rightarrow t\rangle}(\tau_0)\ ,\end{aligned}$$

which cannot be the case by assumption. □

Corollary 4.56. *While backward approximate CP search runs, we know that $\overline{C}[v](\tau) \geq \mathrm{Cost}_G(v,t,\tau)$ is always true for all $v \in V$ and all $\tau \in \mathbb{R}$.*

We are now ready to prove that backward approximate CP search computes the desired node labels.

Lemma 4.57. *After backward approximate CP search terminates, we can be sure that $\underline{C}[v](\tau) \leq \mathrm{Cost}_G(v,t,\tau) \leq \overline{C}[v](\tau)$ holds for all $\tau \in \mathbb{R}$ and all $v \in V$.*

Proof. The proof is analogous to the proof of Lemma 4.21 using Corollary 4.56 in the end. Note that the search direction is backward this time. Also note that the argument requires suffix-optimality instead of prefix-optimality, which is provided in case of all MC paths. □

We not only want to be sure that backward approximate CP search computes correct lower and upper bounds, but also correct predecessor information. This follows directly from Lemma 4.57 together with the second statement of Lemma 4.55.

Corollary 4.58. *After backward approximate CP search terminates, $(G^\top(p))^\top$ contains a (v,t,τ)-MC-path for all $\tau \in \mathbb{R}$ and all $v \in V$ from those t is reachable.*

As in case of most Dijkstra-like algorithms in this thesis, the predecessor information of a node gets permanent as soon as $Q.min()$ gets large enough. Again, this relies on the monotony of $Q.min()$ (see Lemma 4.54).

Lemma 4.59. *The predecessor information $p[v]$ of v does not change anymore as soon as $Q.min() \geq \max \overline{C}[v]$ is fulfilled. From that moment on $(G^\top(p))^\top$ is guaranteed to contain a (v,t,τ)-MC-path for all $\tau \in \mathbb{R}$.*

Proof. The argument is analogous to the proof of Lemma 4.23, in the end using Corollary 4.58. □

Not only the predecessor information of a node v gets permanent when $Q.min()$ reaches $\max \overline{C}[v]$. The same holds true for the node label.

Lemma 4.60. *The label* $\big(\underline{C}[v], \overline{C}[v]\big)$ *of* v *does not change anymore as soon as* $Q.min() \geq \max \overline{C}[v]$ *is fulfilled. So,* $\underline{C}[v](\tau) \leq \mathrm{Cost}_G(v,t,\tau) \leq \overline{C}[v](\tau)$ *is guaranteed for all* $\tau \in \mathbb{R}$ *afterwards.*

Proof. The argument is analogous to the proof of Lemma 4.24. □

4.4 Multi-Label Searches

This section describes the two time-dependent multi-label search algorithms utilized in this thesis. The first one is *time-dependent multi-label search* (see Section 4.4.1). It is a generalization of the basic multi-label search [48, 60] recapitulated in Section 2.2.6. The second one is *time-dependent multi-label A* search* (see Section 4.4.2). It adds goal-direction in the manner of A* search [49], which we recapitulate in Section 2.2.5, to time-dependent multi-label search to achieve faster running times if one-to-one queries have to be answered.

Both algorithms compute MC paths in time-dependent road networks with additional time-invariant costs. In this setup it is not obvious how Dijkstra-like single label searches can be applied successfully—at least if they run in forward direction. The problem is that prefix-optimal MC paths are not guaranteed to exist (see Section 3.2.1 on page 99). Multi-label search, where enough non-optimal paths are kept to guarantee that even a non-prefix-optimal MC path can be found, is a possible solution. Another solution might be backward search (see Section 4.3.5, 4.3.6, and 4.3.7), but it has obvious drawbacks for queries with fixed departure time where the arrival time is unknown.

Note that we also consider how time-dependent multi-label search behaves in the presence of time-dependent additional costs (i.e., edge weights are taken from $\mathscr{F}_\Pi^1 \times \mathscr{X}_\Pi$). This is necessary because time-dependent multi-label search is applied to heuristic TCH structures, where such more general edge weights are allowed (see Section 6.3). In case of time-dependent multi-label A* search we only consider time-invariant additional costs (i.e., edge weights are taken from $\mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$).

4.4.1 Time-Dependent Multi-Label Search

Given a start node $s \in V$ and a departure time $\tau_0 \in \mathbb{R}$, *time-dependent multi label search* (see Algorithm 4.10) computes $\mathrm{Cost}_G(s,u,\tau_0)$ for all $u \in V$ as well as an (s,u,τ_0)-MC-path for all reachable nodes u. It is obtained by applying the idea of original multi-label search [48, 60], which deals with two-dimensional constant travel costs, to time-dependent road networks with additional costs in a straightforward way. Every node $u \in V$ has multiple node labels of the form $(i,u,\tau|\gamma,i')$

assigned where $i \in \mathbb{N}$ is the unique id of the node label itself and $i' \in \mathbb{N}$ the unique id of the preceding node label. Every node label represents a path from the start node s to the respective node u. The path represented by a label $(i, u, \tau|\gamma, i')$ can be extracted iteratively by repeatedly looking up the preceding node label (see Algorithm 2.3 in Section 2.2.6).

The two-dimensional cost $\tau|\gamma$ of a node label $(i, u, \tau|\gamma, i')$ consists of the arrival time τ and the additional cost γ with respect to the path represented by this node label. More precisely, if P_i is the path represented by $(i, u, \tau|\gamma, i')$ and we travel along P_i departing from s at time τ_0, then we arrive in the node u at time τ with additional cost γ; that is, $\tau = \mathbf{arr}\, f_{P_i}(\tau_0)$ and $\gamma = c_{P_i}$. The total travel cost of the path P_i for departure time τ_0 is $C_{P_i}(\tau_0) = f_{P_i}(\tau_0) + c_{P_i} = \tau - \tau_0 + \gamma$. Note that c_{P_i} is independent from the departure time τ_0 (i.e., constant) if the underlying road network only has time-invariant additional costs. But later in this thesis (see Section 6.3) time-dependent multi label search is applied to heuristic TCH structures that can contain more general edge weights; that is, there are edges $u \to_{f|c} v \in E$ with $f|c \in \mathscr{F}_\Pi^1 \times \mathscr{K}_\Pi$. If this is the case, then f is a TTF and c an ACF, where both f and c have points of discontinuity. In such a setup, time-dependent multi-label label search only finds some path from s to t—and not necessarily an MC path. We discuss that at the end of this section.

The PQ key of a node label $(i, u, \tau|\gamma, i')$ is $\tau + \gamma$. Using the pair $\tau|\gamma$ as key, where the underlying order is lexicographic, is not uncommon [60] but slower in the context of road networks in our experience. When a label $(i, u, \tau|\gamma, i')$ of a node u is removed from the PQ (see Line 8), all outgoing edges of u are relaxed. Relaxing an outgoing edge $u \to_{f|c} v \in E$ means that a new label $(j, v, \mathbf{arr}\, f(\tau)|\gamma + c, i)$ is created, where j is the next available label id (see Line 10). The new label is attached to the node v, but only if it is not dominated by another label $(j', v, \sigma|\delta, j'')$ of v. Any node label of v dominated by the new label is removed from v and also from the PQ if it is contained in the PQ. To store the current non-dominated labels of a node u, a label set $L[u]$ is maintained for every node u. Attaching a label to u means adding it to $L[u]$ and removing it from u means removing it from $L[u]$. Obviously, all labels contained in $L[u]$ at the same time are Pareto optimal (see Section 2.1.3) with respect to arrival time and additional cost. Time-dependent multi label search, in contrast to Dijkstra-like single label searches, does not maintain any separate predecessor information. This is not necessary because the predecessor information is already stored within the node labels itself.

After termination, the label set $L[u]$ of every node u reachable from s contains a label $(i_u, u, \tau_u|\gamma_u, i'_u)$, such that $\tau_u + \gamma_u - \tau_0 = \mathrm{Cost}(s, u, \tau_0)$ holds true. The path

$$P_{su} := \mathit{extractPathFromLabelId}(i_u) \subseteq G(L)$$

extracted using Algorithm 2.3 (see Section 2.2.6) is an (s, u, τ_0)-MC-path, if the

Algorithm 4.10. Computes all paths from s to every reachable node $u \in V$ that are Pareto optimal with respect to arrival time and additional cost, although paths with the same arrival time and additional costs are ruled out. The paths to each node u are represented by the node labels in the set $L[u]$. After termination, an (s,u,τ_0)-MC-path is amongst the computed Pareto optimal paths from s to u for every reachable node u—if the additional costs are time-invariant in the underlying road network G. A node label $(i,u,\tau|\gamma,i')$ representing an MC path encodes the corresponding total minimum travel cost; that is, $\mathrm{Cost}_G(s,u,\tau_0) = \tau + \gamma - \tau_0$.

```
function tdMultiLabelSearch($s : V,\ \tau_0 : \mathbb{R}$) : Set
    $L[u] := \emptyset$ for all $u \in V$
    $L[s] := \{(0,s,\tau_0|0,\perp)\}$
    $Q := \emptyset$ : PriorityQueue
    $Q.insert\big((0,s,\tau_0|0,\perp),\tau_0\big)$
    $i_{\text{next}} := 1$                                   // the next unused label id
    while $Q \neq \emptyset$ do
        $(i,u,\tau|\gamma,i') := Q.deleteMin()$
        foreach $u \rightarrow_{f|c} v \in E$ do                // relax outgoing edges of u
            $\ell_{\text{new}} := (i_{\text{next}},\ v,\ \mathbf{arr}\,f(\tau)|\gamma+c,\ i)$
            if $\ell_{\text{new}}$ is not dominated by any label in $L[v]$ then
                $Q.insert(\ell_{\text{new}},\ \mathbf{arr}\,f(\tau)+\gamma+c)$
                remove all labels dominated by $\ell_{\text{new}}$ from $L[v]$ and $Q$
                add $\ell_{\text{new}}$ to $L[u]$
                $i_{\text{next}} := i_{\text{next}} + 1$
```

underlying road network G has time-invariant additional costs. In the presence of time-dependent additional costs, we cannot guarantee that MC paths are found.

Correctness with Time-Invariant Additional Costs. We show that the label set $L[v]$ that time-dependent multi-label search computes for each node $v \in V$ contains an "optimal" label; that is, a label $(i,v,\tau_v|\gamma_v,i')$ with $\tau_v + \gamma_v - \tau_0 = \mathrm{Cost}_G(s,v,\tau_0)$. There, we require that the TTFs fulfill the FIFO property and that the additional costs are time-invariant (i.e., $f|c \in \mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$ for all $u \rightarrow_{f|c} v \in E$). What happens for more general edge weights is discussed later in this section.

Pareto optimal paths (see Section 2.1.3 for a short summary) are vital for the correctness of time-dependent multi-label search. This well-known concept [48, 60] applies in the context of time-dependent road networks with additional time-invariant costs in a straightforward manner. A path $P = \langle s \rightarrow \cdots \rightarrow t\rangle \subseteq G$ is called Pareto optimal for departure time τ_0 if no path R from s to t exists in G, such that

$$f_R(\tau_0) \le f_P(\tau_0) \;\wedge\; c_R \le c_P \;\wedge\; f_R(\tau_0) + c_R < f_P(\tau_0) + c_P \tag{4.17}$$

is fulfilled; that is, no other path from s to t in G strictly dominates P.

Corollary 4.61. *Every (s,t,τ_0)-MC-path $P \subseteq G$ with $s,t \in V, \tau_0 \in \mathbb{R}$ is Pareto optimal for departure time τ_0.*

Corollary 4.61 follows directly from the definition of travel cost as sum of travel time and additional cost.

In Section 3.2.1 we learn that the existence of prefix-optimal MC paths is not guaranteed in time-dependent road networks with additional time-invariant costs. However, a weaker form prefix-optimality concerning Pareto optimal paths is surely provided. Consider a path $P = \langle v_1 \to \cdots \to v_k \rangle \subseteq G$ that is Pareto optimal for departure time τ_0. Then, P is called *Pareto prefix-optimal* for departure time τ_0 if all its prefix paths $\langle v_1 \to \cdots \to v_i \rangle \subseteq P$ with $1 \le i \le k$ are Pareto optimal for departure time τ_0. The following Lemma, which guaranties the existence of Pareto prefix-optimal MC paths, is crucial for the correctness of time-dependent multi-label search.

Lemma 4.62. *For all $s,t \in V, \tau_0 \in \mathbb{R}$ there is a Pareto prefix-optimal (s,t,τ_0)-MC-path in G; if all additional costs in G are constant.*

Proof. Analogous to the proof of Lemma 3.10 but replacing prefixes that are not Pareto optimal by Pareto optimal ones. □

The rest of the correctness proof follows in principle the same pattern as in case of the other Dijkstra-like algorithms considered in this thesis. We begin with the typical monotonous behavior.

Lemma 4.63. *Let $(i_1, v_1, \tau_1 | \gamma_1, i'_1), (i_2, v_2, \tau_2 | \gamma_2, i'_2), \ldots$ be the order in that the node labels are removed from the PQ during time-dependent multi-label search. Then, $\tau_1 + \gamma_1 \le \tau_2 + \gamma_2 \le \ldots$ holds.*

Proof. One can argue by induction over the number of removals from the PQ analogous to Lemma 4.5. The only difference is that labels are removed instead of nodes. □

We continue with the typical statement that node labels and paths correspond. This time, however, the statement is simpler than in case of several approximate single-label searches (compare Lemma 4.19 in Section 4.2.4 for example). This is because every label directly corresponds to one single path.

Lemma 4.64. *During time-dependent multi-label search runs, the path $P_i = \langle s \to \cdots \to v\rangle$ represented by a label $(i, v, \tau_i|\gamma_i, i') \in L[v]$ fulfills $\mathbf{arr}\, f_{P_i}(\tau_0) = \tau_i$ and $c_{P_i} = \gamma_i$, which implies $\tau_i + \gamma_i - \tau_0 = C_{P_i}(\tau_0)$, for all reached nodes $v \in V$.*

Proof. The statement follows easily by induction over the number of hops of a path represented by a label, similar to the proof of Lemma 4.1. □

Just like basic multi-label search, time-dependent multi-label search finds all possible two-dimensional not strictly dominated travel costs in G.

Lemma 4.65. *Time-dependent multi-label search finds all Pareto prefix-optimal paths from s to all $v \in V$ in G, only ruling out paths with the same two-dimensional cost; if all additional costs in G are constant.*

Proof. Assume a Pareto prefix-optimal (s, v_0, τ_0)-MC-path $P_0 = \langle s \to \cdots \to v_0\rangle \subseteq G$ such that no label $(\cdot, v_0, \mathbf{arr}\, f_{P_0}(\tau_0)|c_{P_0}, \cdot)$ is present in $L[v_0]$. Choose a prefix path $P_w := \langle s \to \cdots \to u \to_{f|c} w\rangle \subseteq P_0$ such that $(\cdot, w, \mathbf{arr}\, f_{P_w}(\tau_0)|c_{P_w}, \cdot) \notin L[w]$ but $(\cdot, u, \mathbf{arr}\, f_{P_u}(\tau_0)|c_{P_u}, \cdot) \in L[u]$ with $P_u := \langle s \to \cdots \to u\rangle$. This means, no label $(\cdot, w, \mathbf{arr}\, f(\mathbf{arr}\, f_{P_u}(\tau_0))|c_{P_u} + c, \cdot)$ has been added to $L[w]$ when $u \to_{f|c} w$ was relaxed after the removal of $(\cdot, u, \mathbf{arr}\, f_{P_u}(\tau_0)|c_{P_u}, \cdot)$. This implies a label $\ell_w = (\cdot, w, \tau_w|\gamma_w, \cdot)$ is already present in $L[w]$ at that time with $\tau_w \le \mathbf{arr}\, f(\mathbf{arr}\, f_{P_u}(\tau_0)) = \mathbf{arr}\, f_{P_w}(\tau_0)$ and $\gamma_w \le c_{P_u} + c = c_{P_w}$. But then we have $\tau_w = \mathbf{arr}\, f_{P_w}(\tau_0)$ and $\gamma_w = c_{P_w}$, because P_w is a Pareto optimal path—a contradiction. □

The Pareto prefix-optimality of MC paths implies that an MC path has been found from s to all reachable nodes after termination.

Corollary 4.66. *After time-dependent multi-label search terminates, $L[v]$ contains a label $(i, v, \tau_i|\gamma_i, i')$ representing a path $P_i := \langle s \to \cdots \to v\rangle$ with*

$$\tau_i + \gamma_i - \tau_0 = C_{P_i}(\tau_0) = \mathrm{Cost}(s, v, \tau_0)$$

for all reachable $v \in V$; if all additional costs in G are constant.

Due to its monotonous behavior (see Lemma 4.63), time-dependent multi-label search has found an MC path to a node v as soon as the first label of v is settled.

Lemma 4.67. *The first label $(i, v, \tau_v|\gamma_v, i')$ of a node v that is removed from the PQ during time-dependent multi-label search represents an (s, v, τ_0)-MC-path in G; if all additional costs in G are constant.*

Proof. One can argue analogous to the proof of Lemma 4.6. □

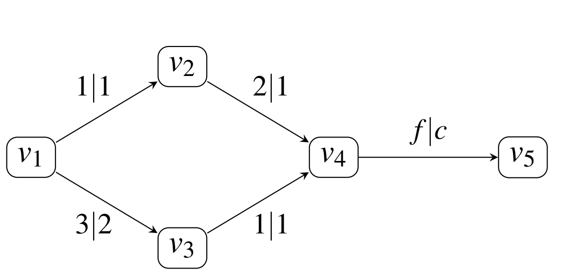
Running Time. For every reachable node, time-dependent multi-label search computes all node labels that are Pareto optimal with respect to arrival time and additional costs. According to Hansen [48] this can be exponentially many in the size of the underlying road network (we recapitulate this in Section 2.1.3). All these labels have to be processed and trigger edge relaxations resulting in the creation of further labels. Whenever we relax an edge $u \to v$, we have to check whether the newly created label is dominated by another label that already exists in $L[v]$ and whether a label in $L[v]$ is dominated by the new one. All this takes additional running time, and even more running time the more Pareto optimal paths exist in G, because $L[v]$ is expected to contain more node labels then.

So, time-dependent multi-label search can take lots of time. This is supported by the fact that answering one-to-one MC queries is NP-hard (see Section 6.1.1). However, the more correlated travel time and additional costs are, the more paths are likely to be dominated by other paths. For this reason we expect longer running time if travel times and additional costs are less correlated.

One-to-One Queries. If we want to compute the cost for traveling from s to t in a road network G where all additional costs in G are constant, then time-dependent multi-label search can be stopped as soon a label $(i, t, \tau_t | \gamma_t, i')$ of t is taken out of the PQ for the first time (see Lemma 4.67). This label represents an (s, t, τ_0)-MC-path and can be extracted from the label id i using Algorithm 2.3 (see Lemma 4.64). If we are only interested in the travel cost, it can be obtained by calculating $\mathrm{Cost}_G(s, t, \tau_0) = \tau_t + \gamma_t - \tau_0$.

TTFs and ACFs with Points of Discontinuity. If edge weights are taken from $\mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$ instead of $\mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$, then time-dependent multi-label search must be adapted a little. This is because the additional cost c of an edge $u \to_{f|c} v$ is no longer a constant, but a function in general. So, c must be replaced by $c(\tau)$ in Line 10 and 12 of Algorithm 4.10. The shortest path structure of the resulting more general time-dependent road networks can be relatively inconvenient because the FIFO property is no longer guaranteed, neither with respect to TTFs nor with respect to TCFs. As a result, MC paths may contain cycles, at least if waiting is forbidden.

Note that we have not proven so far that MC paths always exist in this setup, although we strongly conjecture that this is the case (see Section 3.2.4). Further note that an existing MC path is not necessarily Pareto prefix-optimal. It is even possible that no path at all from a node s to a node t is Pareto-prefix-optimal for some departure time $\tau_0 \in \mathbb{R}$. Figure 4.7 shows a simple example. In such situations, time-dependent multi-label search cannot guarantee to compute an MC path of course. It can not even guarantee to compute Pareto optimal paths.



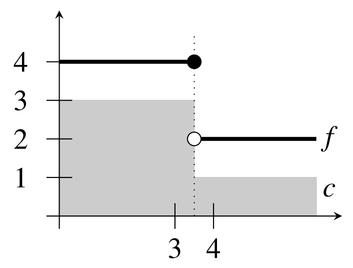


Figure 4.7. A simple example graph that has an edge with piecewise constant (i.e., time-dependent) additional cost. All edges have constant TTFs and ACFs except for the edge $v_4 \to_{f|c} v_5$ with $f|c \in \mathscr{F}_\Pi^1 \times \mathscr{X}_\Pi$ as depicted on the right, where f (thick black line) and c (solid gray) are both discontinuous for departure time 3.5. No path from v_1 to v_5 is Pareto prefix-optimal for departure time 0: $f_{\langle v_1 \to v_3 \to v_4 \to v_5 \rangle}(0)|c_{\langle v_1 \to v_3 \to v_4 \to v_5 \rangle}(0) = 6|4$ dominates $f_{\langle v_1 \to v_2 \to v_4 \to v_5 \rangle}(0)|c_{\langle v_1 \to v_2 \to v_4 \to v_5 \rangle}(0) = 7|5$, but $f_{\langle v_1 \to v_3 \to v_4 \rangle}(0)|c_{\langle v_1 \to v_3 \to v_4 \rangle}(0) = 4|3$ is dominated by $f_{\langle v_1 \to v_2 \to v_4 \rangle}(0)|c_{\langle v_1 \to v_2 \to v_4 \rangle}(0) = 3|2$.

However, if one or more paths exist that are no MC paths but Pareto prefix-optimal, then time-dependent multi-label search finds all these non-optimal Pareto prefix-optimal paths (again ruling out paths with the same two-dimensional travel cost). The one-to-one version of the algorithm finds the cheapest such paths. The reason is that the proofs of Lemma 4.63, 4.64 and 4.65 do not require that the FIFO property is fulfilled.

Corollary 4.68. *If we run time-dependent multi-label search on G with edge weights from $\mathscr{F}_\Pi^1 \times \mathscr{X}_\Pi$, then we find all Pareto prefix-optimal paths from s to all $v \in V$ for departure time τ_0 in G; though paths with the same two-dimensional cost are ruled out.*

Corollary 4.69. *If we run time-dependent multi-label search on G with edge weights from $\mathscr{F}_\Pi^1 \times \mathscr{X}_\Pi$, then the first label $(i, v, \tau_v|\gamma_v, i')$ of a node v removed from the PQ represents a cheapest Pareto prefix-optimal path from s to v in G; if such a path at all exists in G.*

Such a cheapest Pareto prefix-optimal path is not necessarily an MC path.

It is interesting to note how Berger et al. [12] as well as Berger and Müller-Hannemann [13] deal with the lacking Pareto prefix-optimality. Their idea is, essentially, to partition the set of paths into classes of “comparable” paths, and only comparable paths are allowed to dominate each other. Non-comparable paths are kept, even if one them dominates others. Unfortunately, it is not clear how the set of paths could be partitioned to recover the Pareto prefix-optimality here.

4.4.2 Time-Dependent Multi-Label A* Search

Time-dependent multi-label search (see Section 4.4.1) can take lots of running time. On time-dependent road networks with time-invariant additional costs (i.e., edge weights are taken from $\mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$) the running time is not feasible in our experience. To make time-dependent multi-label search run faster, we apply goal direction in the manner of the well-known A* star search [49] (see Section 2.2.5 for a short summary). This is heavily inspired by the multi-objective A* search algorithm NAMOA [59] that applies the idea of A* search to Dijkstra-like multi objective search [60] and settles labels instead of nodes. The basic idea behind A* search is to speed up one-to-one queries using a potential function that estimates the remaining travel costs to the destination node. In this thesis, time-dependent multi-label A* search is used as a reference method to obtain exact query results, which are needed to evaluate the quality of MC querying with heuristic TCHs (see Section 6.4). Note that we apply time-dependent multi-label A* search only to time-dependent road networks with time-invariant additional costs, in contrast to plain time-dependent multi-label search, which we also apply if the additional costs are time-dependent.

Given a start node s, a destination node t, and a departure time τ_0, *time-dependent multi-label A* search* (see Algorithm 4.11) computes $\mathrm{Cost}_G(s,t,\tau_0)$ and a corresponding MC path. The algorithm is in principle the same as the one-to-one version of the plain time-dependent multi-label search (see Algorithm 4.10). The only difference is that we use $\tau + \gamma + \pi_t(v)$ as PQ key of a label $(\cdot, v, \tau|\gamma, \cdot)$ instead of $\tau + \gamma$. The function $\pi_t : V \to \mathbb{R}_{\geq 0}$ is the *potential function* that estimates the travel cost from every node to the destination node t. It must fulfill the condition

$$\pi_t(u) \leq f(\tau) + c + \pi_t(v) \quad \text{for all } u \to_{f|c} v \in E, \tau \in \mathbb{R}\,. \tag{4.18}$$

Note that the potential function π_t as defined in this thesis is relatively restricted. In a more expressive framework, π_t would not only depend on the node but also on the departure time. Ohshima et al. [67] describe this for time-dependent travel times with FIFO property, but without additional costs.

Correctness. We only discuss the case that additional costs are time-invariant (i.e., edge weights are taken from $\mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$). To show that the returned path is an (s,t,τ_0)-MC-path, we argue analogously to the usual correctness argument that is used for original A* search (see Section 10.7 and 10.8 in the textbook by Mehlhorn and Sanders [61] for example); namely, that time-dependent multi-label A* search is nothing but a plain time-dependent multi label search with transformed edge weights (i.e., travel costs) that leave the optimal paths unchanged.

Algorithm 4.11. Time-dependent version of multi-label A*-search. Given start node s, destination node t, and a departure time τ_0, this algorithm computes an (s,t,τ_0)-MC-path. The only differences to time-dependent multi-label search (see Algorithm 4.10) are that the PQ key is adjusted using the potential function π_t and that the algorithm terminates as soon as a label of t is removed from the PQ. To extract the resulting MC path, Algorithm 2.3 is invoked as a subroutine.

```
function tdMultiLabelAStarSearch$(s,t : V,\ \tau_0 : \mathbb{R})$ : Path
    $L[u] := \emptyset$ for all $u \in V$
    $L[s] := \{(0, s, \tau_0|0, \perp)\}$
    $Q := \emptyset$ : PriorityQueue
    $Q.insert\big((0, s, \tau_0|0, \perp),\ \tau_0 + \pi_t(s)\big)$
    $i_{\text{next}} := 1$                                   // the next unused label id
    while $Q \neq \emptyset$ do
        $(i, u, \tau|\gamma, i') := Q.deleteMin()$
        if $u = t$ then return extractPathFromLabelId$(i)$      // see Algorithm 2.3
        foreach $u \to_{f|c} v \in E$ do                        // relax outgoing edges of u
            $\ell_{\text{new}} := (i_{\text{next}},\ v,\ \mathbf{arr}\, f(\tau)|\gamma + c,\ i)$
            if $\ell$ is not dominated by any label in $L[v]$ then
                $Q.insert(\ell_{\text{new}},\ \mathbf{arr}\, f(\tau) + \gamma + c + \pi_t(v))$
                remove all labels dominated by $\ell_{\text{new}}$ from $L[v]$ and $Q$
                add $\ell_{\text{new}}$ to $L[u]$
                $i_{\text{next}} := i_{\text{next}} + 1$
```

To do so, we have to remember that a pair $f|c \in \mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$ with TTF f and additional time-invariant cost c is only a convenient representation of $f|C$ with TTF f and TCF $C = f + c$. To obtain the transformed travel costs used by a plain time-dependent multi-label search that is equivalent to a time-dependent multi-label A* search, we replace the TCF C of an edge $u \to_{f|C} v$ (with $C = f + c$) by the TCF C' defined by

$$C' := f + c + \pi_t(v) - \pi_t(u)\ . \tag{4.19}$$

It is important to note that the transformed TCFs can never be negative due to Equation (4.18).

Lemma 4.70. *MC paths do not change if all edge TCFs are transformed as in Equation* (4.19).

Proof. The transformed travel cost C'_P of a path $P = \langle u_1 \to_{f_1|c_1} \cdots \to_{f_{k-1}|c_{k-1}} u_k \rangle \subseteq$

G amounts to

$$\begin{aligned}C'_P(\tau) &= \sum_{i=1}^{k-1} \big(f_i + c_i + \pi_t(u_{i+1}) - \pi_t(u_i)\big) \circ \mathbf{arr}(f_{i-1} * \cdots * f_1)(\tau) \\ &= \sum_{i=1}^{k-1} f_i \circ \mathbf{arr}(f_{i-1} * \cdots * f_1)(\tau) + \sum_{i=1}^{k-1} \big(c_i + \pi_t(u_{i+1}) - \pi_t(u_i)\big) \ .\end{aligned}$$

with departure time $\tau \in \mathbb{R}$. Considering the second sum in the second line, we see that the values of the potential function telescope along path P, just like in case of original A* search (see Equation (2.30)). We hence obtain

$$\begin{aligned}C'_P(\tau) &= f_{k-1} * \cdots * f_1(\tau) + c_{k-1} + \cdots + c_1 + \pi_t(u_k) - \pi_t(u_1) \\ &= C_P(\tau) + \pi_t(u_k) - \pi_t(u_1) \ .\end{aligned}$$

So, as in case of original A* search, the transformation of a paths travel cost only depends on the paths start and destination node and not on the path in between. This means MC paths stay unchanged (this would be more complicated if π_t would depend on the departure time of course). □

Now we show that time-dependent multi-label A* search simulates the operation of plain time-dependent multi-label search with transformed costs.

Lemma 4.71. *Time-dependent multi-label A* search is equivalent to plain time-dependent multi-label search with transformed edge TCFs as defined by Equation* (4.19).

Proof. We argue by induction over the number k of removals from the PQ that the plain version and the A* version of time-dependent multi-label search perform. This way we show that both searches remove equivalent node labels in the same order, each creating equivalent new node labels with equivalent PQ keys in the same order. Here, "equivalent" means that additional time-invariant costs of equivalent labels of a node v differ by $\pi_t(s) - \pi_t(v)$, which only depends on v, and that all keys differ by the constant value $\pi_t(s)$.

For $k = 1$, the plain version of the multi-label search that runs with transformed costs removes the label $(0, s, \tau_0 | 0, \bot)$ from the PQ and creates a label $(\cdot, v, \mathbf{arr}\, f(\tau_0) | c + \pi_t(v) - \pi_t(s), 0)$ for every relaxed edge $s \rightarrow_{f|c} v \in E$, putting it into the PQ with key $\mathbf{arr}\, f(\tau_0) + c + \pi_t(v) - \pi_t(s)$ each. The A* version equivalently removes the label $(0, s, \tau_0 | 0, \bot)$ from the PQ and for every relaxed edge $s \rightarrow_{f|c} v \in E$ it creates a new label $(\cdot, v, \mathbf{arr}\, f(\tau_0) | c, 0)$, putting it into the PQ with key $\mathbf{arr}\, f(\tau_0) + c + \pi_t(v)$ each. The additional time-invariant costs stored in the labels differ by $\pi_t(s) - \pi_t(v)$ and the keys by $\pi_t(s)$ each. The total costs of the initial labels of s differ by $\pi_t(s) - \pi_t(s) = 0$.

For $k > 1$, the plain and the A* version both perform their k-th removal taking the equivalent labels $(i, u, \tau | \gamma + \pi_t(u) - \pi_t(s), i')$ and $(i, u, \tau | \gamma, i')$ out of the PQ respectively. Then, the relaxation of an edge $u \rightarrow_{f|c} v$ creates the new labels

$$\begin{aligned}&\big(\cdot, u, \mathbf{arr} f(\tau) \big| \gamma + \pi_t(u) - \pi_t(s) + c + \pi_t(v) - \pi_t(u), i\big)\\ &\qquad\qquad = \big(\cdot, u, \mathbf{arr} f(\tau) \big| \gamma + c + \pi_t(v) - \pi_t(s), i\big)\end{aligned}$$

and $(i, u, \mathbf{arr} f(\tau) | \gamma + c, i)$, respectively, which are obviously equivalent (i.e., they differ by $\pi_t(s) - \pi_t(v)$). The respective keys are $\mathbf{arr} f(\tau) + \gamma + c + \pi_t(v) - \pi_t(s)$ and $\mathbf{arr} f(\tau) + \gamma + c + \pi_t(v)$ and differ by $\pi_t(s)$. □

Together with Lemma 4.67 this yields the desired correctness statement.

Corollary 4.72. *The path returned by time-dependent multi-label A* search is an (s, t, τ_0)-MC-path and the corresponding label $(\cdot, t, \tau_t | \gamma_t, \cdot)$ fulfills*

$$\tau_t + \gamma_t - \tau_0 = \mathrm{Cost}_G(s, t, \tau_0) \ ;$$

if all additional costs are constant.

Running Time. The running time of time-dependent multi-label A* search depends on the choice of the potential function π_t. Two extreme cases are

- that $\pi_t(u) = 0$ holds for all $u \in V$, which means time-dependent multi-label A* search degenerates into a plain time-dependent multi-label search, and
- that $\pi_t(u, \tau) = \mathrm{Cost}(u, t, \tau)$ holds for all $u \in V, \tau \in \mathbb{R}$, which means time-dependent multi-label A* search only removes labels from the PQ that belong to an (s, t, τ_0)-MC path. This would require π_t not only to depends on the node but also on the time, which we do not consider in this thesis.

For $\pi_t(u)$ lying somewhere between 0 and $\mathrm{Cost}_G(u, t, \tau)$, less labels are processed if $\pi_t(u)$ lies closer to $\mathrm{Cost}_G(u, t, \tau)$.

Lemma 4.73. *Consider the potential functions $\pi_t, \pi'_t : V \rightarrow \mathbb{R}$. Let M and M' be the set of labels removed from the PQ if π_t and π'_t are used respectively. Then, we have*

$$\forall u \in V : \mathrm{Cost}_G(u, t, \tau_0) \geq \pi'_t(u) > \pi_t(u) \geq 0 \quad \Longrightarrow \quad M' \subseteq M \ .$$

Proof. First, we argue $\pi_t(t) = \pi'_t(t) = 0$ because of $\mathrm{Cost}_G(t, t, \tau_0) = 0$. This means the first label removed from the PQ is of the form $(\cdot, t, \tau_t | \gamma_t, \cdot)$, regarless whether the potential function π_t or π'_t is used. Second, assume a label $\ell_x = (\cdot, x, \tau_x | \gamma_x, \cdot) \in M' \setminus M$ of a node x. The label ℓ_x is removed from the PQ before the first label of t

is removed, but only if the potential function π'_t is used. With π_t, in contrast, it is removed after the first label of the node t is removed. This implies

$$\tau_t + \gamma_t \leq \tau_x + \gamma_x + \pi_t(x) < \tau_x + \gamma_x + \pi'_t(x) \leq \tau_t + \gamma_t \ ,$$

a contradiction. □

Note that $M' = M$ is possible. As a simple example assume that G only consists of a path from s to t. It is interesting that the slightly weaker condition $\mathrm{Cost}_G(u,t,\tau_0) \geq \pi'_t(u) \geq \pi_t(u) \geq 0$ allows the existence of labels $\ell_x \in M' \setminus M$.

A statement analogous to Lemma 4.73 has been made in the context of original A* search (i.e., A* search for one-dimensional constant travel costs) by Goldberg and Harrelson [47], but without proof. They mention that the condition $\pi'_t(u) > \pi_t(u)$ can be weakened to $\pi'_t(u) \geq \pi_t(u)$ if the node id is utilized to break ties with respect to the PQ key. In the context of multi-label A* search, an analogous argument can be applied. But then, the label id must be utilized instead of the node id for tie breaking.

A Simple Potential Function. A simple example of a potential function is

$$\pi_t(u) := \underline{\mathrm{Cost}}_G(u,t) \ .$$

To show that $\underline{\mathrm{Cost}}_G(\cdot,t)$ fulfills the condition in Equation (4.18), we choose an arbitrary edge $u \rightarrow_{f|c} v \in E$ and argue

$$\begin{aligned}\underline{\mathrm{Cost}}_G(u,t) &= \min\Big\{ \sum_{i=1}^{k-1} \min C_i \,\Big|\, \langle u \rightarrow_{f_1|c_1} \cdots \rightarrow_{f_k|c_k} t \rangle \subseteq G \Big\} \\ &\leq \min f + c + \min\Big\{ \sum_{i=1}^{\ell-1} \min C_i \,\Big|\, \langle v \rightarrow_{f_1|c_1} \cdots \rightarrow_{f_\ell|c_\ell} t \rangle \subseteq G \Big\} \\ &\leq f(\tau) + c + \underline{\mathrm{Cost}}_G(v,t)\end{aligned}$$

for all $\tau \in \mathbb{R}$ with $C_i := f_i + c_i$. To compute $\pi_t = \underline{\mathrm{Cost}}_G(\cdot,t)$ we could run Dijkstra's algorithm in a backward manner starting from t before the time-dependent multi-label A* search starts. But then we would process the whole graph as we would not know when to stop. Instead we perform a backward CP interval search that computes node labels $[q[u], r[u]] = [\underline{\mathrm{Cost}}_G(u,t), \overline{\mathrm{Cost}}_G(u,t)]$ and can be stopped as soon as $Q.min()$ exceeds $q[s]$. This is possible, because the transpose predecessor graph $(G^\top(p))^\top$ of backward CP interval search already contains an (s,t,τ)-MC-path for all $\tau \in \mathbb{R}$ at that time (see Corollary 4.53).

Also, time-dependent multi-label A* search can use $r[u]$ to maintain an upper bound of the desired result $\mathrm{Cost}_G(s,t,\tau_0)$ and then refrain from relaxing edges

$u \rightarrow_{f|c} v$ where $(\tau_u + \gamma_u) + (f(\tau_u) + c) + \pi_t(v)$ with $(\cdot, u, \tau_u|\gamma_u, \cdot) \in L[u]$ exceeds this upper bound. Another idea is not to run time-dependent multi-label A* search in G but to restrict it to the transpose predecessor graph $(G^\top(p))^\top$ of backward CP interval search, which contains all relevant paths from s to t.

If the potential function $\pi_t = \underline{\mathrm{Cost}}_G(\cdot, t)$ works well, then the overall running time is the running time of one-to-one backward CP interval search plus the moderate running time of a time-dependent multi-label A* search that does not process too much labels. The running time is similar to the running time of Dijkstra's algorithm in this case; which is not very fast but good enough for a reference method used to obtain exact query results. However, if travel times and additional costs are not so related, then the number of processed labels can nevertheless explode leading to impractical running times; and in our experience this really happens.

Changing to a time-dependent potential function $\pi_t : V \times \mathbb{R} \rightarrow \mathbb{R}_{\geq 0}$ may further reduce the number of processed node labels. There, $\pi_t(u, \cdot)$ is a CP estimating the remaining travel cost to t depending on the node u. The CPs $\pi_t(u, \cdot)$ could be computed by a preceding backward approximate CP search for example. This takes, of course, more time than backward CP interval search but makes the multi-label A* search faster. This yields a tradeoff between the accuracy of the CPs and the number of processed node labels. However, we have not tried this so far.

4.5 References

This chapter is heavily based on a journal article published together with Geisberger, Sanders, and Vetter [8] and a conference article published together with Geisberger, Neubauer, and Sanders [6]. It adopts all the descriptions of basic Dijkstra-like algorithm from these articles and compiles them into one chapter, adding several details here and there. Many wordings of these articles have been used or rephrased. The proofs of correctness in this chapter, which are a lot of material, have mostly been worked out newly.

The time-dependent Dijkstra (see Dreyfus [35]) described in Section 4.2.1 and the travel time profile (TTP) search (see Orda and Rom [69]) described in Section 4.2.2 are well established. Note that the formulation of TTP search by Orda and Rom is rather similar to the well-known Bellmann-Ford algorithm, in contrast to ours, which is similar to today's typical formulation of Dijkstra's algorithm with PQ (i.e., "Dijkstra-like").

5
Minimizing Time-Dependent Travel Times

This chapter describes how CHs [44] can be generalized to deal with the case that travel costs are *time-dependent travel times*. Remember that CHs are an algorithmic framework originally designed to provide fast and exact route planning for constant travel costs (see Section 2.3 for an introduction). The *time-dependent CHs (TCHs)* considered in this chapter are able to answer exact one-to-one EA and TTP queries efficiently in time and space. The more general case of time-dependent travel costs beyond travel times is treated in Chapter 6.

The original CHs, which are designed for constant travel costs, have some crucial properties to assure that one-to-one queries are answered both fast and exact. To provide fast and exact answering of one-to-one EA and TTP queries, TCHs must fulfill analogous properties. This results in a list of requirements (see Section 5.1). TCHs work in two stages, just like the original CHs: the preprocessing stage and the querying stage. CH preprocessing is already quite costly for constant travel costs. If travel costs are time-dependent travel times, however, things get even more complicated. It is, in fact, not at all trivial to provide a TCH preprocessing that terminates in reasonable time (see Section 5.2). In case of original CHs, one-to-one queries can be answered relatively easily. One only has to perform a bidirectional Dijkstra search that goes upward in the hierarchy. For EA queries, however, one must overcome the problem that backward search must be performed without knowing the arrival time. For TTP queries this is not a problem, but one has to deal with the expensive operations on TTFs there (see Section 5.3).

TCH structures allow fast and exact answering of EA and TTP queries, but need lots of space. The reason is that TTFs associated with shortcut edges tend to have lots of bend points. However, the information stored in the TTFs of shortcut edges is actually redundant. For this reason it is still possible to provide fast and exact answering of EA and TTP queries, even if we only store approximate TTFs in case of shortcuts. The resulting *approximate TCH (ATCH)* structures need far less space than exact ones. In case of EA queries one has to accept a moderate slowdown. In case of TTP queries one even gains an additional speedup (see Section 5.4). Accepting a small error, TTP queries can be answered even faster using the similarly space efficient *inexact TCHs*. For EA queries, inexact TCHs are far less interesting because ATCHs already provide fast and exact answering of EA queries with small memory requirements. Nevertheless, EA queries are possible with inexact TCHs, too (see Section 5.5).

An experimental evaluation confirms that TCHs really accomplish what we promise above: TCHs and ATCHs provide fast and exact EA and TTP querying. ATCHs considerably save space, while EA queries get only moderately slower and TTP queries even get faster. Inexact TCHs also save space while showing small errors and small running times. Especially, they provide very fast but inexact TTP querying (see Section 5.6).

5.1 Requirements

To provide fast and exact route planning with time-dependent travel times as costs, TCH structures have to fulfill a number of requirements adapted from original constant travel cost CHs [44].

5.1.1 Requirements to Make TCHs Exact

A CH structure H constructed from a road network $G = (V,E)$ with constant travel costs [44] (see Section 2.3 for a short summary) fulfills the following very important properties:

- $G \subseteq H$ with $\mu_H(s,t) = \mu_G(s,t)$ for all $s,t \in V$ (see Lemma 2.7),
- if $t \in V$ is reachable from s in G, then H contains a shortest up-down-path from s to t in H (see Lemma 2.8), and
- every up-down-path in H can be expanded recursively to a path of the same total cost in G (see Corollary 2.9).

These three properties ensure the exactness of CHs, because bidirectional upward search is guaranteed to find a shortest up-down-path that can be expanded to a shortest path in G. To ensure the exactness of TCHs, three analogous properties must be fulfilled.

First, a TCH structure H must contain the original road network G and the EA times in H must be the same as in G. Otherwise, H could not be used to compute EA paths in G. Theorem 5.4 confirms that the TCHs described in this chapter fulfill this requirement (see Section 5.2.2).

Second, a TCH structure H must guarantee the existence of prefix-optimal EA up-down-paths. A path $\langle s \to \cdots \to x \to \cdots \to t\rangle \subseteq H$ is called an *up-down-path* if its prefix path $\langle s \to \cdots \to x\rangle$ only leads upward (i.e., $s \prec \cdots \prec x$), and its suffix path $\langle x \to \cdots \to t\rangle$ only leads downward (i.e., $t \prec \cdots \prec x$). The node x is called *top node*. If an up-down-path $\langle s \to \cdots \to x \to \cdots \to t\rangle$ is also an (s,t,τ_0)-EA-path, then it is called an *earliest arrival (EA) up-down-path* for departure time τ_0 (or, an (s,t,τ_0)-EA up-down-path for short). To ensure that bidirectional upward search can be used to answer one-to-one EA and TTP queries, H must contain an (s,t,τ_0)-EA up-down-path for all $s,t \in V$ and all $\tau_0 \in \mathbb{R}$ (provided that t is reachable from s). Theorem 5.5 confirms that the TCHs described in this chapter fulfill this requirement (see Section 5.2.2). Note that Theorem 5.5 even guarantees the existence of *prefix-optimal* EA-up-down paths. This makes it possible to apply the pruning technique stall-on-demand—which is adopted from original CHs—in the context of TCHs (for details see Section 5.3.3).

Third, a TCH structure H must guarantee that a shortcut $u \to_f v$ can be expanded to a path $P_\tau = \langle u \to \cdots \to v\rangle \subseteq G$ for every departure time $\tau \in \mathbb{R}$, such that $f(\tau) = f_{P_\tau}(\tau)$ holds true. Note that the expansion can be different for different departure times. Annotating the shortcut $u \to_f v$ with a single middle node, as in case of constant travel costs [44], is obviously not enough to achieve such a behavior. Instead, we annotate every shortcut with a *shortcut descriptor* storing multiple middle nodes that are valid for different departure intervals (for details see Section 5.2.1). The shortcut descriptors are enough to enable the required time-dependent expansion of shortcut edges (see Corollary 5.7).

5.1.2 Requirements to make TCHs Fast

The original constant travel costs CHs [44] are not only an exact, but also a fast route planning technique. To achieve this, CH structures must be flat and sparse (as already mentioned in Section 1.1.4 and 2.3). We call a CH flat if the maximum number of hops $m_\uparrow$ or $m_\downarrow$ that an upward or downward path has, respectively, is small. We call a CH sparse if the out-degree of nodes is not too large in $H_\uparrow$ and $H_\downarrow$. If $m_\uparrow$ and $m_\downarrow$ as well as the out-degrees are small, then the bidirectional Dijkstra search reaches the top of the CH quite soon while relaxing not too many edges. This results in small search spaces, which implies small query times. EA and TTP queries, which are also based on bidirectional upward search, should profit from flat and sparse hierarchies, too.

5.2 Preprocessing

A TCH structure is constructed by successively contracting the nodes of G with respect to a given node order $\prec$. Section 5.2.1 describes what node contraction means if travel costs are time-dependent travel times. Section 5.2.2 explains the complete construction process and how it can be finished within reasonable time. Constructing a TCH requires that the importance relation $\prec$ is already known. But if this is not the case, the node order has to be determined first. Node ordering as we do it, however, is basically a construction process with a lot of additional work. The details are can be found in Section 5.2.3. It must be noted that node ordering not only yields the importance relation $\prec$ but also a corresponding TCH structure. A further construction step is not necessary then. However, a node order once computed can be reused to perform the construction of TCHs for other sets of travel costs if these are not too different. We examine this in our experiments (see Section 5.6). Note that both node ordering and TCH construction parallelize quite naturally on shared memory architectures (see Section 5.2.4).

Note that Christian Vetter made crucial contributions to the preprocessing during a student research project [82] and as a student assistant, both regarding concepts and implementation. This includes the parallelization of the preprocessing as well as very effective optimizations that make the preprocessing much more faster (see Section 5.7 for details).

5.2.1 Contracting a Node

Node contraction is the basic operation that creates the next higher level of the hierarchy from the current one. More precisely, the contraction of a node x in G yields a *time-dependent* overlay graph $G' := (V', E')$ of G with $V' = V \setminus \{x\} \subseteq V$ and $\mathrm{EA}_{G'}(s,t,\tau) = \mathrm{EA}_G(s,t,\tau)$ for all $s,t \in V', \tau \in \mathbb{R}$. This generalizes the definition of overlay graphs by Holzer et al. [51] to work with time-dependent travel times.

Necessary Shortcuts. In principle, contracting a node x means to remove x and all its incident edges from the graph without changing the EA times between the remaining nodes. We achieve this by replacing each path $\langle u \to_f x \to_g v \rangle$ by a shortcut edge $u \to_{g*f} v$ when necessary. A shortcut is considered to be *necessary* if $\langle u \to_f x \to_g v \rangle$ is an EA path for some departure time; that is, if

$$\exists \tau \in \mathbb{R} : \mathrm{EA}_G(u,v,\tau) = \mathbf{arr}(g * f)(\tau) \tag{5.1}$$

holds. When a shortcut is necessary, an edge $u \to_h v$ may already be present. Then, we do not insert another edge but *merge* the edges: We replace $u \to_h v$ by

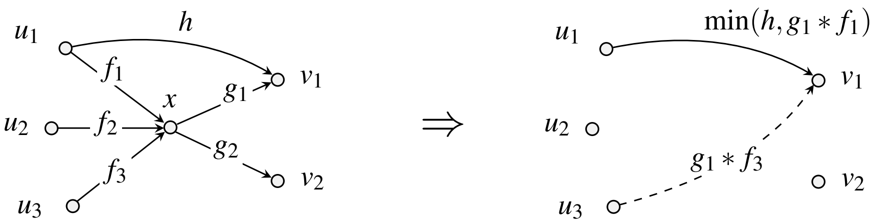


Figure 5.1. Illustration of what happens when a node x is contracted. The situation before and after contraction is depicted on the left and the right respectively. On the right, x and all its incident edges are removed. Assume $\langle u_1 \rightarrow_{f_1} x \rightarrow_{g_1} v_1 \rangle$ and $\langle u_3 \rightarrow_{f_3} x \rightarrow_{g_1} v_1 \rangle$ are the only removed paths that are EA paths for some departure time each. So, the shortcuts $u_1 \rightarrow_{g_1 * f_1} v_1$ and $u_3 \rightarrow_{g_4 * f_1} v_1$ are necessary and put into E_x. An edge $u_1 \rightarrow_h v_1$ is already present and merged with $u_1 \rightarrow_{g_1 * f_1} v_1$ resulting in the TTF $\min(h, g_1 * f_1)$. The other shortcut $u_3 \rightarrow_{f_3 * g_1} v_1$ is newly inserted (drawn dashed).

$u \rightarrow_{\min(g*f,h)} v$ avoiding parallel edges this way. This is different from constant travel costs, where we simply replace parallel edges with greater travel costs by edges with smaller ones. The reason is that, in the time-dependent case, both parallel edges can be EA paths but for different departure times.

We write $\overline{G} := (\overline{V}, \overline{E}) := (V \setminus \{x\}, \{u \rightarrow v \in E \,|\, u, v \neq x\})$ to denote what remains of G when x and its incident edges are removed. Then,

$$\begin{aligned} E_x := \big\{ u \rightarrow_{g*f} v \;\big|\; & \langle u \rightarrow_f x \rightarrow_g v \rangle \subseteq G \text{ and} \\ & \exists \tau \in \mathbb{R} : \mathbf{arr}(g * f)(\tau) = \mathrm{EA}_G(u, v, \tau) \big\} \end{aligned} \tag{5.2}$$

is the set of necessary shortcuts when $x \in V$ is contracted. If parallel edges are merged as described above, then the contraction of the node x yields an overlay graph $G' := (V \setminus \{x\}, E')$ with edge set

$$\begin{aligned} E' := & \big\{ u \rightarrow_h v \in \overline{E} \;\big|\; u \rightarrow v \notin E_x \big\} \cup \\ & \big\{ u \rightarrow_{g*f} v \in E_x \;\big|\; u \rightarrow v \notin \overline{E} \big\} \cup \\ & \big\{ u \rightarrow_{\min(h, g*f)} v \;\big|\; u \rightarrow_h v \in \overline{E} \text{ and } u \rightarrow_{g*f} v \in E_x \big\} \; . \end{aligned} \tag{5.3}$$

Figure 5.1 illustrates what is happening when a node x is contracted. Before the contraction of x we have the graph G, after the contraction of x the graph G'.

Lemma 5.1. *Contracting a node x in a graph G preserves all EA times between the remaining nodes. That is, $\mathrm{EA}_{G'}(s, t, \tau) = \mathrm{EA}_G(s, t, \tau)$ for all $s, t \in V \setminus \{x\}$.*

Proof. Consider a path $P' = \langle s \rightarrow \cdots \rightarrow t \rangle \subseteq G'$ and a departure time $\tau_0 \in \mathbb{R}$. Let $u \rightarrow_h v$ be an edge of P' with $u \rightarrow_{g*f} v \in E_x$. Then we have

$$f_{P'}(\tau_0) = f_{\langle v \rightarrow \cdots \rightarrow t \rangle} * h * f_{\langle s \rightarrow \cdots \rightarrow u \rangle}(\tau_0) \; .$$

We write $\tau_u := \mathbf{arr}\, f_{\langle s \to \cdots \to u \rangle}(\tau_0)$. There are two possible cases: If $h(\tau_u) = g * f(\tau_u)$, then we replace $u \to_h v$ in P' by $\langle u \to_f x \to_g v \rangle \subseteq G$. Otherwise, there is an edge $u \to_{h_{\text{old}}} v$ in $\overline{G} \subseteq G$ with $h(\tau_u) = h_{\text{old}}(\tau_u)$, and we replace h in $u \to_h v$ by h_{old}. Replacing all such edges transforms P' into a path $P_0 \subseteq G$ with $f_{P_0}(\tau_0) = f_{P'}(\tau_0)$. Altogether, we obtain $\text{EA}_{G'}(s,t,\tau) \geq \text{EA}_G(s,t,\tau)$ for all $s,t \in V$ and all $\tau \in \mathbb{R}$, because τ_0 has been chosen arbitrarily from $\mathbb{R}$.

Now, consider a path $P = \langle s \to \cdots \to t \rangle \subseteq G$ and a departure time $\tau_0 \in \mathbb{R}$. If P contains a subpath $\langle u \to_f x \to_g v \rangle$, then we either have $g * f(\tau_u) = \text{EA}_G(u,v,\tau_u)$ or $g * f(\tau_u) > \text{EA}_G(u,v,\tau_u)$, again with $\tau_u := \mathbf{arr}\, f_{\langle s \to \cdots \to u \rangle}(\tau_0)$.

In case $g * f(\tau_u) = \text{EA}_G(u,v,\tau_u)$, we further distinguish whether $u \to_{h_{\text{old}}} v \in \overline{E}$ exists or not. For $u \to_{h_{\text{old}}} v \in \overline{E}$, we replace the subpath $\langle u \to_f x \to_g v \rangle$ of P by $u \to_{\min(h_{\text{old}}, g*f)} v \in E'$. For $u \to_{h_{\text{old}}} v \notin \overline{E}$, we replace $\langle u \to_f x \to_g v \rangle$ by $u \to_{g*f} v \in E'$. Anyway, the subpath $\langle u \to_f x \to_g v \rangle$ of P is replaced by an edge $u \to_h v$ with $h(\tau_u) \leq g * f(\tau_u)$ and the resulting path has no later arrival time because of

$$\begin{aligned}\mathbf{arr}\, f_P(\tau_0) &= \mathbf{arr}\, f_{\langle v \to \cdots \to t \rangle}\big(\mathbf{arr}(g*f)(\tau_u)\big) \\ &\geq \mathbf{arr}\, f_{\langle v \to \cdots \to t \rangle}\big(\mathbf{arr}\, h(\tau_u)\big)\ .\end{aligned}$$

The above inequality holds because $f_{\langle v \to \cdots \to t \rangle}$ fulfills the FIFO property.

In case $g * f(\tau_u) > \text{EA}_G(u,v,\tau_u)$, we consider a prefix-optimal (u,v,τ_u)-EA-path $P_{uv} = \langle u \to \cdots \to v \rangle \subseteq G$, which exists due to Lemma 3.10. Replacing $\langle u \to_f x \to_g v \rangle$ by P_{uv} in P does again not increase the arrival time because of

$$\begin{aligned}\mathbf{arr}\, f_P(\tau_0) &= \mathbf{arr}\, f_{\langle v \to \cdots \to t \rangle}\big(\mathbf{arr}(g*f)(\tau_u)\big) \\ &\geq \mathbf{arr}\, f_{\langle v \to \cdots \to t \rangle}\big(\mathbf{arr}\, f_{P_{uv}}(\tau_u)\big)\ .\end{aligned}$$

This way, we obtain a new version of P that contains P_{uv} and has no later arrival time. If $P_{uv} \not\subseteq \overline{G}$ holds (i.e., P_{uv} goes via the node x) we go back to the already considered case that a subpath $\langle u \to_f x \to_g v \rangle$ of P fulfills $g * f(\tau_u) = \text{EA}_G(u,v,\tau_u)$, which is the case because P_{uv} is prefix-optimal.

It can further happen that P contains an edge $u \to_{h_{\text{old}}} v \in \overline{E}$ with $u \to_{g*f} v \in E_x$. We replace every such edge by $u \to_{\min(h_{\text{old}}, g*f)} v \in E'$. In the end, we obtain a path R' from s to t in G' with $f_{R'}(\tau_0) \leq f_P(\tau_0)$. But τ_0 has been chosen arbitrarily from $\mathbb{R}$. So, $\text{EA}_{G'}(s,t,\tau) \leq \text{EA}_G(s,t,\tau)$ holds for all $s,t \in V$ and all $\tau \in \mathbb{R}$. □

Time-Dependent Shortcuts. A shortcut $u \to_{g*f} v \in E_x$ (see Equation (5.2)) is an artificial edge that has no corresponding single road segment in the real world. Instead, it represents the path $\langle u \to_f x \to_g v \rangle$; that is, the two-hop path leading over the middle node x. In case of constant travel costs, every shortcut is annotated with a single middle node (as in case of original CHs [44], see Section 2.3.1). With the middle nodes, every shortcut edge $u \to v$ can be expanded recursively to the original path $\langle u \to \cdots \to v\rangle \subseteq G$ it represents (see Section 2.3.2). This important idea is also used if the travel costs are time-dependent.

However, things are more complicated in case of time-dependent travel costs because the two-hop path represented by a shortcut may change over time. More precisely, a *time-dependent shortcut* can represent different two-hop paths for different departure times. The reason is that necessary shortcuts $u \to_{g*f} v$ are merged with edges $u \to_h v$ if present. If $u \to_h v$ is a shortcut itself, with middle node y, then $u \to_{\min(h,g*f)} v$ may have two middle nodes: middle node x for departure times τ with $g * f(\tau) < h(\tau)$ and middle node y for $g * f(\tau) > h(\tau)$. For $g * f(\tau) = h(\tau)$, we are free to choose x or y as middle node.

We hence annotate every shortcut with multiple middle nodes, each such middle node x together with a subset $A_x \subseteq \mathbb{R}$ containing departure times where the middle node x is valid. Note that an edge $u \to_h v$ may sometimes act as a shortcut and sometimes as an original edge (i.e., as an edge of the original road network G). The latter can be expressed easily by simply annotating the symbol $\perp$ instead of a middle node, together with a departure set $A_\perp \subseteq \mathbb{R}$. Thanks to an observation made by Foschini et al. [36], we can be sure EA paths exist that are valid on compact departure intervals.

Observation 5.2 ([36]). *Consider $s,t \in V$. The interval $[0,\Pi]$ can be partitioned into k compact intervals $[a_0,a_1],[a_1,a_2]\ldots,[a_{k-1},a_k]$ (with $a_0 = 0$, $a_k = \Pi$), such that $P_1,\ldots,P_k$ exist where each P_i is an (s,t,τ)-EA-path in G for all $\tau \in [a_{i-1},a_i]$.*

So, as all involved TTFs are periodic, it is enough to annotate every time-dependent shortcut $u \to_{\min(h,g*f)} v \in E'$ with a finite sequence of intervals and corresponding middle nodes; that is,

$$\big\langle([a_0,a_1],x_1),([a_1,a_2],x_2),\ldots,([a_{k-1},a_k],x_k)\big\rangle \tag{5.4}$$

with $0 = a_0 < a_1 < \cdots < a_k = \Pi$ and $x_1,\ldots,x_k \in V \cup \{\perp\}$. With this information at hand, a shortcut can be expanded depending on the departure time.

We call such a sequence as in Equation (5.4) a *shortcut descriptor*. The right middle node for a given departure time τ_0 can be found in $\mathrm{O}(\log k)$ time using binary search. But according to our experience, k tends to be quite small in practice. So, linear scanning is most probably faster due to caching effects.

Merging of Shortcut Descriptors. Consider the two parallel edges $u \to_f v$ and $u \to_g v$ with the shortcut descriptors $\langle([a_0,a_1],x_1),\ldots,([a_{k-1},a_k],x_k)\rangle$ and $\langle([b_0,b_1],y_1),\ldots,([b_{\ell-1},b_\ell],y_\ell)\rangle$ respectively. To compute a valid shortcut descriptor of the merged edge $u \to_{\min(f,g)} v$, we modify Algorithm 3.2 to find out for which departure times f or g is smaller. More precisely, we compute a minimum

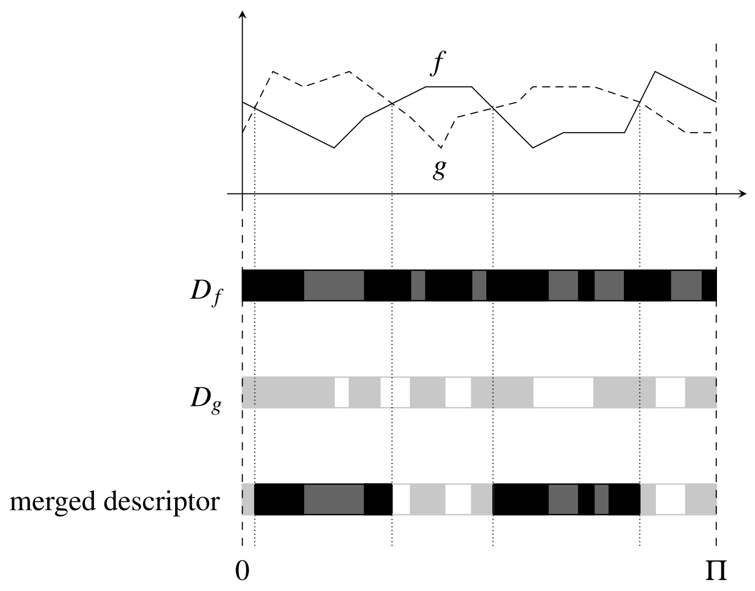


Figure 5.2. Illustration how merging of shortcut descriptors works. Assume that two shortcuts $u \rightarrow_f v$ and $u \rightarrow_g v$ with TTFs f (drawn solid) and g (drawn dashed) as well as shortcut descriptors D_f (black/dark gray) and D_g (white/light gray), respectively, are merged. The merged shortcut descriptor equals the shortcut descriptor D_f for departure times where f lies below g. For departure times where g lies below f, the merged shortcut descriptor equals D_g. The period of the TTFs is Π.

length sequence

$$\big\langle([c_0,c_1],h_1),([c_1,c_2],h_2),\dots,([c_{n-1},c_n],h_n)\big\rangle \tag{5.5}$$

with $0 = c_0 < c_1 < \cdots < c_n = \Pi$ and $h_1,\dots,h_n \in \{f,g\}$, such that

- $h_i = f$ implies $f(\tau) \le g(\tau)$ and
- $h_i = g$ implies $g(\tau) \le f(\tau)$.

for all $\tau \in [c_{i-1},c_i]$. Computing this sequence takes $\mathrm{O}(|f|+|g|+n)$ time.

A shortcut descriptor of the merged edge $u \rightarrow_{\min(f,g)} v$ can then be obtained by further refining the partition $[c_0,c_1],[c_1,c_2],\dots,[c_{n-1},c_n]$ of $[0,\Pi]$. To do so, we overlay each interval $[c_{i-1},c_i]$ with the shortcut descriptor of $u \rightarrow_f v$ or $u \rightarrow_g v$ depending on h_i. More precisely, for each $i \in \{1,\dots,n\}$, we build the sequence

$$S_i^{(f)} := \big\langle([c_{i-1},a_{j_i}],x_{j_i}),([a_{j_i},a_{j_i+1}],x_{j_i+1}),\dots,([a_{j_i+n_i-1},c_i],x_{j_i+n_i})\big\rangle$$

with $a_{j_i-1} \le c_{i-1} < a_{j_i}$ and $a_{j_i+n_i-1} < c_i \le a_{j_i+n_i}$ if $h_i = f$, or the sequence

$$S_i^{(g)} := \big\langle([c_{i-1},b_{k_i}],y_{j_i}),([b_{k_i},b_{k_i+1}],y_{k_i+1}),\dots,([b_{k_i+m_i-1},c_i],y_{k_i+m_i})\big\rangle$$

with $b_{k_i-1} \leq c_{i-1} < b_{k_i}$ and $b_{k_i+m_i-1} < c_i \leq b_{k_i+m_i}$ if $h_i = g$. The concatenation

$$S := S_1^{(h_1)} S_2^{(h_2)} \cdots S_n^{(h_n)}$$

is then a valid shortcut descriptor for the merged edge $u \to_{\min(f,g)} v$. Figure 5.2 illustrates how such a *merged shortcut descriptor* arises from the original ones.

Note that the merged shortcut descriptor (i.e., the concatenated sequence S) may contain consecutive pairs with the same middle node. More precisely, S may have the form

$$S = \big\langle \ldots, ([a,b],x), ([b,c],x), \ldots \big\rangle \,.$$

Such pairs can be melted, which means S is transformed to $S = \langle \ldots, ([a,c],x), \ldots \rangle$. It must be noted that merged shortcut descriptors are not unique, because the TTFs f and g may be equal on a closed interval.

The running time of merging shortcut descriptors is always linear in the complexity of the involved TTFs and the size of the involved shortcut descriptors.

Lemma 5.3. *Consider two edges $u \to_f v$, $u \to_g v$ with shortcut descriptors D_f and D_g, respectively. Then, computing a merged shortcut descriptor S of $u \to_{\min(f,g)} v$ takes* $\mathrm{O}(|D_f| + |D_g| + |f| + |g|)$ *time.*

Proof. The sequence of length n in Equation (5.5) can be computed by a straightforward modification of Algorithm 3.2, which takes $\mathrm{O}(|f| + |g| + n)$ time. Computing S as described above takes $\mathrm{O}(|D_f| + |D_g| + n)$ time in total, because simultaneous scanning of the two original shortcut descriptors D_f, D_g and of the sequence in Equation (5.5) suffices. Whether f or g is smaller can only change $\mathrm{O}(|f| + |g| + |\min(f,g)|)$ times, which implies $n = \mathrm{O}(|f| + |g|)$ according to Corollary 3.12. We have a total running time of $\mathrm{O}(|D_f| + |D_g| + |f| + |g|)$ hence. $\square$

5.2.2 Constructing a TCH Structure

A TCH construction is a sequence of node contractions (see Section 5.2.1) in the order given by $\prec$; that is, in the order $x_1 \prec \cdots \prec x_{|V|}$. The result of this process (i.e., a TCH structure) is a hierarchy of time-dependent overlay graphs

$$G_1 = (V_1, E_1), \ldots, G_{|V|} = (V_{|V|}, E_{|V|}) \tag{5.6}$$

with $V_{i+1} = V_i \setminus \{x_i\}$. For constant travel costs, however, we already explained that CH structures are never represented this way (see Section 2.3.1). This would need too much memory. Instead, we just store the original graph G together with all inserted and merged shortcuts. This results in the graph $H := (V, E_H)$ with

$$\begin{aligned} E_H := \Big\{ u \to_f v \;\Big|\; & u \to v \in \textstyle\bigcup_{i=1}^{|V|} E_i \text{ and} \\ & f : \tau \mapsto \min\big\{ g(\tau) \;\big|\; u \to_g v \in E_i \text{ for some } i \big\} \Big\}\,, \end{aligned} \tag{5.7}$$

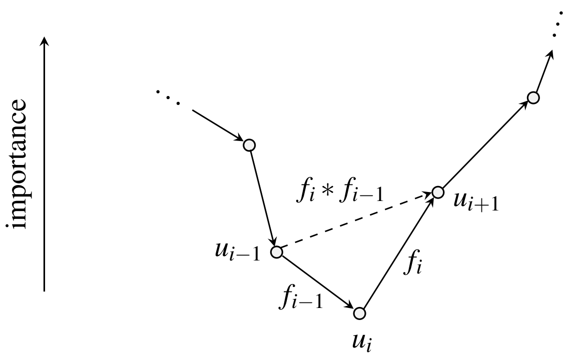


Figure 5.3. Illustration of the main argument used in the proof of Theorem 5.5. We look at the TCH structure H "from the side" with the more important nodes higher up. The node u_i is a local minimum on the prefix-optimal path P. Because of the prefix-optimality, a shortcut $u_{i-1} \rightarrow_{f_i * f_{i-1}} u_{i+1}$ (drawn dashed) must have been considered as necessary by the preprocessing and must hence be present in H (though possibly merged with other original edges and shortcut edges).

which is a pretty condensed representation of the hierarchy. We also have to remember whether an edge leads upward or downward with respect to the relation $\prec$. Together with the final merged shortcut descriptors created during node contraction, this is all information we need for fast and exact EA and TTP queries.

In fact, TCH structures as characterized by Equation (5.7) fulfill the requirements stated in Section 5.1.1. This is confirmed by Theorem 5.4 below, which states that a TCH contains its original graph and has the same EA times, as well as by Theorem 5.5 below, which guarantees the existence of prefix-optimal EA up-down-paths. Moreover, the information stored in the shortcut descriptors is enough to provide time-dependent expansion of shortcuts (see Corollary 5.7).

Theorem 5.4. *Let H be a TCH constructed from $G = (V, E)$. Then, we have $G \subseteq H$ and $\mathrm{EA}_H(s,t,\tau) = \mathrm{EA}_G(s,t,\tau)$ for all $s,t \in V$ and all $\tau \in \mathbb{R}$.*

Proof. We know from Equation (5.7) that H contains every edge $u \rightarrow_f v \in E_1 = E$, though maybe with a different TTF f'. We can be sure, however, that $f'(\tau) \leq f(\tau)$ holds for all $\tau \in \mathbb{R}$. Because of the FIFO property, this implies $\mathrm{EA}_H(s,t,\tau) \leq \mathrm{EA}_G(s,t,\tau)$ for all $s,t \in V$ and all $\tau \in \mathbb{R}$. But $\mathrm{EA}_H(s,t,\tau) < \mathrm{EA}_G(s,t,\tau)$ is not possible, because $G_1, \dots, G_{|V|}$ is a hierarchy of overlay graphs. So, no edge in any of $E_1, \dots, E_V$ enables an earlier arrival time than the edges in E do. □

Theorem 5.5. *Let H be a TCH constructed from $G = (V, E)$. For $s,t \in V$ such that t is reachable from s in G and for $\tau_0 \in \mathbb{R}$, there is an up-down-path in H that is also a prefix-optimal (s,t,τ_0)-EA-path.*

Proof. According to Lemma 3.10 there exists a prefix-optimal (s,t,τ_0)-EA-path $P := \langle u_1 \rightarrow_{f_1} \cdots \rightarrow_{f_{k-1}} u_k \rangle$ in G. According to Theorem 5.4, P is also a prefix-optimal (s,t,τ_0)-EA-path in H. If P is not an up-down-path we choose a *local minimum* of P excluding s and t; that is, we choose $i \in \{2,\dots,k-1\}$ such that $u_i \prec u_{i-1}, u_{i+1}$. By the prefix-optimality of P (and also by its suffix-optimality) we know that $\langle u_{i-1} \rightarrow_{f_{i-1}} u_i \rightarrow_{f_i} u_{i+1} \rangle$ is an $(u_{i-1}, u_{i+1}, \mathrm{EA}_G(s,u_{i-1},\tau_0))$-EA-path in H. So, when contracting u_i during preprocessing, a shortcut $u_{i-1} \rightarrow_{f_i * f_{i-1}} u_{i+1}$ is considered to be necessary (see Section 5.2.1). Figure 5.3 illustrates this argument. This means that an edge $u_{i-1} \rightarrow_h u_{i+1}$ is present in H with

$$\mathbf{arr}\, h\big(\mathrm{EA}_G(s,u_{i-1},\tau_0)\big) \le \mathbf{arr}(f_i * f_{i-1})\big(\mathrm{EA}_G(s,u_{i-1},\tau_0)\big) = \mathrm{EA}_G(s,u_{i+1},\tau_0)\ .$$

This implies $\mathrm{EA}_G(s,u_{i+1},\tau_0) = \mathbf{arr}\, h(\mathrm{EA}_G(s,u_{i-1},\tau_0))$ due to Theorem 5.4 . As a consequence $P' := \langle u_1 \rightarrow_{f_1} \cdots \rightarrow_{f_{i-2}} u_{i-1} \rightarrow_h u_{i+1} \rightarrow_{f_{i+1}} \cdots \rightarrow_{f_{k-1}} u_k \rangle$ is a prefix-optimal (s,t,τ_0)-EA-path in H. By setting $P := P'$ the above argument can be applied again and again until the resulting path P^* is an up-down-path from s to t. Note that this process surely terminates, because P' has one edge less than P. Also, a single edge is clearly an up-down-path. So, by construction, P^* is an up-down-path as well as a prefix-optimal (s,t,τ_0)-EA-path in H. □

The given proof is very similar to the proof by Geisberger et al. [44] that states that CHs work correctly with constant travel costs. We recapitulate this proof in Section 2.3.2 (see the proof of Lemma 2.8).

Like in case of CHs with constant travel costs, a TCH structure H can be decomposed into an upward graph $H_\uparrow$ and a downward graph $H_\downarrow$ defined by

$$\begin{aligned} H_\uparrow &:= (V,E_\uparrow) := (V, \{u \rightarrow_f v \in E_H \mid u \prec v\}) \\ H_\downarrow &:= (V,E_\downarrow) := (V, \{u \rightarrow_f v \in E_H \mid v \prec u\}) \end{aligned} \tag{5.8}$$

with $H = H_\uparrow \cup H_\downarrow$. These graphs are edge disjoint DAGs; that is, $E_\uparrow \cap E_\downarrow = \emptyset$. It is obvious that the upward and the downward part of an up-down-path $\langle s \rightarrow \cdots \rightarrow x \rightarrow \cdots \rightarrow t \rangle \subseteq H$ with top node x completely lies in $H_\uparrow$ and $H_\downarrow$, respectively. More precisely, we have

$$\langle s \rightarrow \cdots \rightarrow x \rangle \subseteq H_\uparrow \quad \text{and} \quad \langle x \rightarrow \cdots \rightarrow t \rangle \subseteq H_\downarrow\ . \tag{5.9}$$

Basic TCH Construction. Conceptually, the TCH construction yields a hierarchy of overlay graphs $G_1,\dots,G_{|V|}$. We already explained, however, that the hierarchy is never represented this way. Instead, the construction yields the condensed representation characterized by Equation (5.7). During construction, the hierarchy of overlay graphs is also never represented explicitly. Instead, we successively construct a TCH H starting from G by inserting and merging more and more shortcuts. Initially, we set $H := G$. All shortcuts $u \to_{g*f} v \in E_x$ that are considered to be necessary when a node x is contracted are added to H (or merged into H if an edge $u \to_h v$ is already present respectively). We also maintain the *remaining graph* $R = (V_R, E_R)$. More precisely, R is the subgraph of H, which is induced by all nodes that are *not* contracted yet.[1] A node x and its incident edges are removed from R as soon as x is contracted. The remaining graph R is an overlay graph of both G and H.

The construction phase is divided in *iterations*. In every iteration we contract a number of nodes. At the beginning of every iteration we build a set

$$I := \big\{x \in V_R \;\big|\; \forall u \in N_R^1(x) : x \prec u\big\} \tag{5.10}$$

of nodes that are less important than all their neighbors in R. Note that I is an independent node set[2] in R. Thanks to this property, it is relatively easy to parallelize the TCH construction (see Section 5.2.4). Having built the set I, we contract all the nodes in I. This includes that all necessary shortcuts are added to H (or merged into H respectively). The remaining graph R has changed now: On the one hand, all nodes in I and their adjacent edges have been removed from R. On the other hand, some new shortcuts may be present in R and some edges in R may have a modified TTF now (due to merging).

The next iteration works exactly like the one before: We build another independent node set I, which is a subset of the now smaller set of remaining nodes V_R. Then, we contract all the nodes in the new set I while inserting some shortcuts into H (or merging some shortcuts respectively). Then, we perform another iteration and so on. We repeat this process until R is the empty graph. When this process ends, all nodes are contracted and H is completely transformed into a TCH structure.

Algorithm 5.1 shows pseudocode describing the construction process. Line 6 selects all nodes in V_R that are locally minimal with respect to the total order $\prec$ and puts them into the set I. These are exactly the nodes specified by Equation (5.10). Line 8 and 11 select the set E_x of shortcuts that are considered to be necessary if a node $x \in I$ is contracted (see Equation (5.2)). In Line 10 the incoming and outgoing edges of x are marked as downward and upward edges respectively.

That nodes are not simply contracted in the order given by $\prec$ but repeatedly collected into an independent set $I \subseteq V$ of local minima with respect to $\prec$ (see Equation (5.10)), is a difference to the construction procedure of original CHs [44] that has been introduced by Vetter when he parallelized the TCH preprocessing during a student research project [82].

[1]The subgraph of a graph $G = (V, E)$ that is *induced* by a subset of nodes $X \subseteq V$ is defined as the subgraph $\big(X, \{u \to v \in E \mid \{u, v\} \subseteq X\}\big) \subseteq G$.

[2]An *independent node set* is a set of nodes in a graph such that no two nodes in the set are adjacent.

Algorithm 5.1. The basic TCH construction procedure. Given a graph (V,E) with time-dependent travel times as travel costs, and also given a node order $\prec$, this algorithm constructs a TCH structure.

```
function constructTch((V,E) : Graph, ≺ : NodeOrder) : Graph
    $E_H := E$                                        // G is subgraph of H
    $(V_R,E_R) := (V,E)$                              // the "remaining graph" $R = (V_R,E_R)$
    annotate all edges in $E_H$ and $E_R$ with shortcut descriptor $\langle([0,\Pi],\bot)\rangle$
    while $E_R \neq \emptyset$ do
        $I := \{x \in V_R \mid \forall u \in N^1_{(V_R,E_R)}(x) : x \prec u\}$    // see Equation (5.10)
        foreach $x \in I$ do
            $E_x := \emptyset$                        // set of necessary shortcuts when contracting x
            foreach path $\langle u \to_f x \to_g v\rangle$ in $(V_R,E_R)$ do
                mark $u \to x$ as downward and $x \to v$ as upward edge in $E_H$
                if $\exists \tau \in \mathbb{R} : g*f(\tau) = \mathrm{TT}_{(V_R,E_R)}(u,v,\tau)$ then add $u \to_{g*f} v$ to $E_x$
            remove $x$ and all its incident edges from $(V_R,E_R)$   // at this point $R = \overline{G}$
            foreach $u \to_{g*f} v \in E_x$ do
                if there is $u \to_h v \in E_R$ then
                    $D :=$ shortcut descriptor of $u \to_h v$
                    $S :=$ merge shortcut descriptors $D$ and $\langle([0,\Pi],x)\rangle$
                    replace $u \to_h v$ by $u \to_{\min(h,g*f)} v$ in $E_H$ and $E_R$
                    annotate $u \to_{\min(h,g*f)} v$ with merged shortcut descriptor $S$
                else
                    add $u \to_{g*f} v$ to $E_H$ and $E_R$
                    annotate $u \to_{g*f} v$ with shortcut descriptor $\langle([0,\Pi],x)\rangle$
            // at this point $R = G'$
    return $(V, E_H)$
```

Checking for Necessary Shortcuts. Line 11 checks whether a shortcut is necessary or not. To do so, one wants to find out, whether some $\tau \in \mathbb{R}$ exists with $g*f(\tau) = \mathrm{TT}_{(V_R,E_R)}(u,v,\tau)$. The condition itself can be checked within

$$\mathrm{O}\big(|g*f| + \big|\mathrm{TT}_R(u,v,\cdot)\big|\big)$$

time in a manner quite similar to computing $\min(g*f, \mathrm{TT}_R(u,v,\cdot))$. It can be done using an adaption of Algorithm 3.2 hence. The computation of the *witness profile* $\mathrm{TT}_R(u,v,\cdot)$, however, is much more difficult. The reason is that this so called *witness search* requires a one-to-one TTP query on the remaining graph R. Answering this TTP query by simply performing the one-to-one version TTP search (see Section 4.2.2) would be so time consuming, that the TCH construction

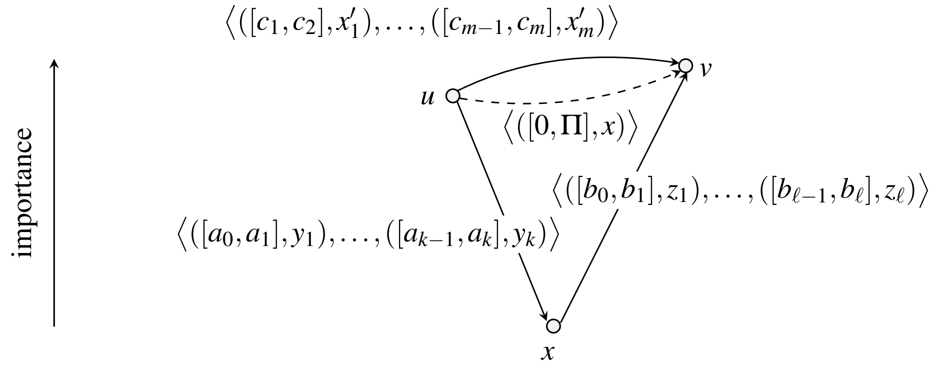

$\langle([c_1,c_2],x'_1),\dots,([c_{m-1},c_m],x'_m)\rangle$
importance
u
v
$\langle([0,\Pi],x)\rangle$
$\langle([b_0,b_1],z_1),\dots,([b_{\ell-1},b_\ell],z_\ell)\rangle$
$\langle([a_0,a_1],y_1),\dots,([a_{k-1},a_k],y_k)\rangle$
x

Figure 5.4. Merging of shortcut descriptors during TCH construction. Like in Figure 5.3, we look at the TCH structure H "from the side" but with the difference that H is not finished jet and still under construction. The newly created shortcut $u \to v$ (drawn dashed) has the simple shortcut descriptor $\langle([0,\Pi],x)\rangle$, which indicates that this shortcut represents the path $\langle u \to x \to v\rangle$ for all departure times. If another edge $u \to v$ (drawn solid) with a shortcut descriptor $\langle([c_0,c_1],x'_1),\dots([c_{m-1},c_m],x'_m)\rangle$ is already present, then this shortcut descriptor has to be merged with the simple shortcut descriptor $\langle([0,\Pi],x)\rangle$. Note that the shortcut descriptors of the edges $u \to x$ and $x \to v$ do *not* contribute to the merged shortcut descriptor.

would not terminate within reasonable time then (as our experiments reported in Section 5.6 show).

Alternatively, one may simply want to add *all* possible shortcuts to E_x, regardless whether they are necessary or not. This is also not good, because the resulting TCH, which is likely to be quite dense, would probably need far too much space and queries would probably be much slower. However, witness search can be optimized to make TCH construction feasible. This is explained a little later in this section.

Merging Shortcuts. Lines 13 to 21 insert and merge the necessary shortcuts into H as specified by Equation (5.3) and also maintain the shortcut descriptors. The shortcut descriptor of a newly inserted shortcut is simply $\langle([0,\Pi],x)\rangle$, as it represents the path $\langle u \to_f x \to_g v\rangle$ for all possible departure times (see Line 21). If an edge $u \to_h v$ is already present, then the new shortcut is not inserted but merged of course (see Lines 14 to 18). This also affects the shortcut descriptor, which has to be merged, too. The simple shortcut descriptor $\langle([0,\Pi],x)\rangle$ that solely represents the path $\langle u \to_f x \to_g v\rangle$ is merged with the shortcut descriptor D of the already present edge $u \to_h v$. Figure 5.4 illustrates such a situation.

Note that the merging of a newly created shortcut $u \to_{g*f} v$ with the already present edge $u \to_h v$ can in general not be done in $\mathrm{O}(|h|+|g*f|) = \mathrm{O}(|f|+|g|+|h|)$ time. Though computing $g*f$ and $\min(h, g*f)$ only takes $\mathrm{O}(|h|+|g*f|)$ time (see Corollary 3.11 and 3.12), we cannot apply this upper bound to the merging of shortcut descriptors. The reason is that h may even be constant regardless whether D contains several middle nodes emerging from earlier contractions. In practice, however, we rather suppose $|D| = \mathrm{O}(|h|)$.

On-the-fly Externalization. Once contracted, a node x and its incident edges are no longer needed by the TCH construction. The only reason why it is kept in memory as part of the resulting TCH structure H, is that H serves as a return value when the TCH construction is finished. However, instead of returning H as a whole, one could also externalize H (for example, writing it to disk) on the fly. This is especially easy if the externalized H is represented as an edge list.[3] If this is the case, one simply appends the edges incident to a contracted node to the externalized data. The contracted node and the incident edges can be deleted afterwards.

It must be noted that especially the TTFs of shortcuts higher up in the hierarchy can get quite complex. So, the described on-the-fly externalization of H may save a lot of memory. The TTFs of the deleted edges must also be deleted of course. If the graph data structure does not support the deletion of nodes or edges, then deleting the TTFs only may already save a considerable amount of space.

Optimized Witness Search. Without further optimizations, TCH construction as described in Algorithm 5.1 takes too much time. This is because witness search (i.e., the computation of witness the profile $\mathrm{TT}_R(u,v,\cdot)$) makes TCH construction impractically slow if implemented by a simple TTP search (see Section 4.2.2) on R. To make witness search feasible, we apply five optimizations which we explain in the following. Two of them, the *heuristic corridor thinning* and the *sample search* have been contributed by Vetter when he was working as a student assistant. He also adopted the *hop limit* from original CHs [44] choosing a maximum hop distance of 16.

Preceding TTP Interval Search. One idea to make witness search faster is not about improving the computation of $\mathrm{TT}_R(u,v,\cdot)$, but to avoid this computation if possible. To do so, we perform the one-to-one version of TTP interval search (see Section 4.2.3) on R with start node u and destination node v first, computing the interval $[q[v], r[v]] = \left[\underline{\mathrm{TT}}_R(u,v), \overline{\mathrm{TT}}_R(u,v)\right]$. So, if $r[v] < \min(g * f)$ holds, we know that *no* shortcut is needed—*without* doing TTP search. We call $[q[v], r[v]]$ a *witness interval* in this case. Otherwise, $[q[v], r[v]]$ and $[\min g * f, \max g * f]$ overlap. This means we know nothing and perform TTP search.

[3]An *edge list* is a very simple representation of a graph $G = (V,E)$ as sequence $\langle u_1 \to_{f_1} v_1, \dots, u_{|E|} \to_{f_{|E|}} v_{|E|} \rangle$ with $E = \{u_1 \to_{f_1} v_1, \dots, u_{|E|} \to_{f_{|E|}} v_{|E|}\}$.

Algorithm 5.2. Given two nodes s,t and a predecessor information p, this algorithm computes the corridor from s to t in the predecessor graph represented by p. The computation is actually a BFS starting from t. Every touched edge is added to the resulting corridor (V_S,E_S).

```
function $extractCorridor(s,t : Node, p : PredInfo) : Graph$
    $(V_S,E_S) := (\emptyset,\emptyset)$
    $Q := \emptyset : FifoQueue$
    $Q.push(t)$
    mark $t$ as $visited$
    while $Q \neq \emptyset$ do
        $v := Q.pop()$
        foreach $u \in p[v]$ do
            add $u \to v$ to $E_S$
            if $u$ already marked as $visited$ then continue
            $Q.push(u)$
            mark $u$ as $visited$
    add all nodes that are marked as $visited$ to $V_S$
    return $(V_S,E_S)$
```

Restricting Witness Search to a Corridor. If TTP search is performed after a preceding TTP interval search, as just explained, then several edges can be ignored by the TTP search: Let p be the predecessor information generated by the TTP interval search performed before witness search. Then, we perform the TTP search only on a subgraph of the predecessor graph $G(p) \subseteq R$; namely, the corridor $S := \mathscr{P}_{R(p)}(u,v) \subseteq R(p) \subseteq R$, which exactly consists of all paths from u to v in the predecessor graph $R(p)$. A TTP search restricted to the corridor S, still computes $\mathrm{TT}_R(u,v,\cdot)$ correctly. This is because $\mathrm{TT}_S(u,v,\cdot) = \mathrm{TT}_R(u,v,\cdot)$ holds, which follows from the fact that $R(p)$, and thus S, contains an (u,v,τ)-EA-path for all $\tau \in \mathbb{R}$. This, is due to Lemma 4.14.

The idea behind the restriction to S is to make TTP search a good deal more goal-directed, which means that less nodes are processed and less edges are relaxed. We hope to exclude some nodes that do not to lie on an EA path from u to v this way. Our experiments (see Section 5.6) show that TCH construction runs a good deal faster with the preceding TTP interval search described above and with the restriction to the corridor S. Note that we have not examined how much faster the preprocessing runs with preceding TTP interval search but without restriction to the corridor S.

To obtain the corridor S we only have to perform a BFS[4] in $R(p)^\top$ starting

[4]BFS (breadth-first search) is a fundamental method to traverse a graph starting from a given

from v while adding a transposed version of all touched edges to S. Note that the graph $R^{\top}(p)$ needs not to be build explicitly (it is, in fact, already there). Instead, the BFS search uses the predecessor information set up by the TTP interval search directly. Algorithm 5.2 shows the respective pseudocode. It is invoked by $S := \mathit{extractCorridor}(u, v, p)$, where p is predecessor information generated by the preceding TTP interval search.

Thinning out the Corridor Heuristically. We can further decrease the time spend on TTP search by using an even "thinner" corridor than S; that is, we use a subgraph of S containing less edges. To do so, we do not store the whole the predecessor information during preceding TTP interval search. Instead, we only remember *two* predecessors of each node v with label $[q[v], r[v]]$ at a time: the one that lastly improved $q[v]$ and the one that lastly improved $r[v]$ (i.e., $|p[v]| \leq 2$). In other words, we rather maintain two predecessor nodes $p_q[v], p_r[v] \in V \cup \{\perp\}$ than a set $p[v] \subseteq V$. There, $p_q[v]$ and $p_r[v]$ are updated together with $q[v]$ and $r[v]$ respectively. If the relaxation of an edge $u \rightarrow_f v$ decreases $q[v]$, for example, then $p_q[v]$ is updated by assigning $p_q[v] := u$. The other predecessor node $p_r[v]$ is maintained analogously. Of course, the resulting thinner corridor

$$S' := \mathscr{P}_{R(p_q) \cup R(p_r)}(u, v) \subseteq S \subseteq R$$

does in general not contain all EA paths from u to v. It may even contain no EA path for some departure times; that is, departure times $\tau_0 \in \mathbb{R}$ may exist with

$$\mathrm{EA}_{S'}(u, v, \tau_0) > \mathrm{EA}_S(u, v, \tau_0) = \mathrm{EA}_R(u, v, \tau_0) = \mathrm{EA}_G(u, v, \tau_0) \; .$$

As a consequence, TTP search when performed on S' no longer computes $\mathrm{TT}_R(u, v, \cdot)$ but $\mathrm{TT}_{S'}(u, v, \cdot)$, which may be greater for the one or another departure time; that is, $\tau \in \mathbb{R}$ may exist with $\mathrm{TT}_{S'}(u, v, \tau) > \mathrm{TT}_R(u, v, \tau)$. This may lead to unnecessary shortcuts as we may fail to prove that a shortcut is not needed. But this happens not too often, which gets apparent from the fact that the resulting TCH structures are sufficiently sparse to provide fast querying. Moreover, the running time spend on TTP search is reduced greatly (our experiments confirm this, see Section 5.6). So, thinning out the corridor is a heuristic that works very well. It is also a conservative heuristic, as it *never* prevents any necessary shortcut.

Note that the corridor thinning has been contributed by Vetter when he was working as a student assistant.

Sample Search. Another idea to further reduce the cases where witness search is performed is *sample search*, a one-to-one time-dependent Dijkstra search on

node (e.g., see the textbook by Mehlhorn and Sanders [61]).

R from u to v with departure time $\Pi/2$. Even if $[q[v], r[v]]$, as computed by the TTP interval search, and $[\min(g*f), \max(g*f)]$ overlap, sample search might still yield $\mathrm{EA}_R(u,v,\Pi/2) = \mathbf{arr}(g*f)(\Pi/2)$. Then, we know that the shortcut is necessary without witness search.

However, we perform sample search only occasionally: We maintain a value β that we set to zero at the very beginning of the construction process. Every time a shortcut is inserted, we increase β by some value λ^+. If a potential shortcut is *not* inserted, we decrease β by another value λ^-. If β gets larger than some threshold ξ, we switch to the *sample search mode*, meaning that we always perform a sample search before TTP interval search. If, in contrast, β gets smaller than $-\xi$, we switch back to the *no sample search mode*. In our implementation we chose $\lambda^+ := 4$, $\lambda^- := 1$, and $\xi := 1\,000$. As an intuition, we perform sample search when shortcuts are more probable, and we omit sample search, when shortcuts are less probable.

Note that the sample search has been contributed by Vetter when he was working as a student assistant.

Hop Limit. To prevent that optimized witness search—which includes a sample search, a TTP interval search, and a TTP search, all starting from the node u—takes too much time, we limit the search radius to k hops. This *hop limit* is adopted from the preprocessing of original CHs with constant travel costs [44]. The idea is that the edges of a node w that is removed from the PQ are *not* relaxed if the EA path from the start node u to w has more than k edges. In our implementation, we use $k = 16$, which has been chosen by Vetter. It must be noted that the predecessor graphs of TTP search and TTP interval search do not contain a unique path from u to w. In these cases, we just use the number of hops that emerge from the edge relaxed last.

The results of TTP interval search may be altered by the hop limit. It may then compute labels $[q[v], r[v]]$ with $q[v] \geq \underline{\mathrm{TT}}_R(u,v)$ and $r[v] \geq \overline{\mathrm{TT}}_R(u,v)$ instead of equality. Then, the check $r[v] < \min(g*f)$ may fail, even if $\overline{\mathrm{TT}}_R(u,v) < \min(g*f)$ holds. Running TTP search on the corridor S or S' also leads to another result then. So, the hop limit may provoke further unnecessary shortcuts, just like thinning out the corridor S does. Our experiments show, however, that the resulting TCH structures are sparse enough to allow small running times of EA and TTP queries (see Section 5.6).

Note that the hop limit has been adopted from the original CHs [44] by Vetter when he was working as a student assistant.

5.2.3 Ordering the Nodes

We already suggested that the node ordering is actually a TCH construction plus a lot of extra work. In every iteration the node ordering chooses an independent node set I in the remaining graph R. Then, it contracts all nodes in I while adding some shortcuts to H or merging them respectively. Having finished an iteration, we choose another independent node set I in the new R and so on. This is the same procedure as TCH construction described in Section 5.2.2 and all the techniques described there are also applied. However, $\prec$ is not yet fully established during this process. So, we need another way to decide which nodes are contracted next than the one characterized by Equation (5.10); that is, another way to decide which nodes are put into I. To do so, we assign a *tentative cost* value $x.cost$ to every node $x \in V$. This way we estimate how attractive a node is to be contracted. The nodes with smaller cost should be contracted earlier.

Actually, the tentative cost $x.cost$ estimates how the remaining graph R would change when the node x were contracted. Mainly, this means the number of edges that would be removed and inserted respectively (though other factors are also included). The number of removed and inserted edges, however, depends on the incoming and outgoing edges that x has in R, because every path $\langle u \rightarrow_f x \rightarrow_g v \rangle \subseteq R$ may provoke another necessary shortcut. Accordingly, the tentative cost of a node x must be updated, whenever a neighbor y of x is contracted. This has two reasons: First, x loses the edge $x \rightarrow y$, the edge $y \rightarrow x$, or both when y is contracted. Second, x may gain additional edges. This can happen, for example, if a shortcut $x \rightarrow z$ representing a path $\langle x \rightarrow y \rightarrow z \rangle$ is necessary when y is contracted.

To keep the cost $x.cost$ of a node x up-to-date, we perform a *simulated contraction* of x whenever x gains or loses an edge in R. This happens exactly when a neighbor y of x in R is *really* contracted. After the simulated contraction of x we update $x.cost$. Note that the simulated contraction does not alter H or R. Instead, we compute E_x (see Equation (5.2)) solely to update $x.cost$.

Basic Node Ordering. Algorithm 5.3 shows pseudocode for the node ordering procedure. Though node ordering procedure is a TCH construction (see Algorithm 5.1) plus lots of extra work, the pseudocode looks very different. The reason is, that we use a much more compact representation here, which is more convenient but much less detailed than in Algorithm 5.1. The treatment of shortcut descriptors is even omitted completely. The extra work that is done by the node ordering happens in Lines 4 to 7, 10, and 17. Without these changes, Algorithm 5.3 would describe the same procedure as Algorithm 5.1. The only exception is the set I, which is set up in a different way (see Line 9).

In an initial step, we determine the initial tentative cost for all nodes by performing a simulated contraction for every node $u \in V$ (see Lines 6 and 7). Then

Algorithm 5.3. Computes a node order $\prec$ and a corresponding TCH structure $H = (V, E_H)$ for a given graph $G = (V, E)$. The pseudocode looks very different from Algorithm 5.1, although it is in principle the same procedure plus lots of additional work. This is because we use a more abstract notation here. The returned node order $\prec$ is simply represented as a sequence of nodes. The most time consuming part of the algorithm is at the end of the main loop where the simulated contractions are performed. The treatment of shortcut descriptors is omitted, but Algorithm 5.1 shows how it is handled.

```
1  function orderNodes((V,E) : Graph) : (Graph, NodeOrder)
2      $E_H := E$                                        // G is subgraph of H
3      $(V_R, E_R) := (V, E)$                             // the "remaining graph" $R = (V_R, E_R)$
4      $\prec := \langle\rangle$                         // the node order represented as sequence of nodes
5      set up random bijective function $noTie : V \to \{1, \dots, |V|\}$
6      foreach $u \in V$ do
7          simulate contraction of $u$ and set $u.cost$
8      while $E_R \neq \emptyset$ do
9          $I := \{x \in V_R \mid \forall u \in N^2_{(V_R,E_R)}(x) \setminus \{x\} : (x.cost < u.cost$ or
                   $(x.cost = u.cost$ and $noTie(x) < noTie(u)))\}$
10         append all $x \in I$ to $\prec$
11         $E_{\text{new}} := \bigcup_{x \in I} E_x$ with $E_x$ as in Equation (5.2)
12         $V_{\text{neighbors}} := N^1_{V_R,E_R}(I)$
13         for all $x \in I$ mark all edges $u \to x \in E_H$ as downward edge
14         for all $x \in I$ mark all edges $x \to v \in E_H$ as upward edge
15         for all $x \in I$ remove $x$ and all its incident edges from $(V_R, E_R)$
16         insert or merge all $u \to_{g*f} v \in E_{\text{new}}$ into $(V_R, E_R)$ and $(V_H, E_H)$
17         simulate contraction of all $u \in V_{\text{neighbors}}$ and update $u.cost$ each
18     return $((V, E_H), \prec)$
```

we perform a sequence of iterations similar to the construction process described in Section 5.2.2. The independent node set, however, is chosen differently this time (see Line 9). Instead of a set I as described in Equation (5.10) we choose $I \subseteq V_R$ with the property

$$x \in I \quad \Longrightarrow \quad \forall u \in N^2_R(x) \setminus \{x\} : \left(u \notin I \text{ and } x.cost \leq u.cost\right), \tag{5.11}$$

which means that every node in I has minimal cost in its 2-hop neighborhood and none of its 2-hop neighbors is also added to I. There, we have the problem that a nodes $y \in N^2_R(x) \setminus \{x\}$ may have the same cost as x; that is, $x.cost = y.cost$. In this case we use the random bijective function $noTie : V \to \{1, \dots, |V|\}$ for tie breaking, which is set up in Line 5.

Note that an independent node set I that fulfills Equation (5.11) is different from an independent node set I as characterized by Equation (5.10). And this is not only the case because we use the tentative cost instead of $\prec$. Especially, the gap between the nodes is at least three hops instead of two hops as in case of Equation (5.10). This is needed by the parallel node ordering in shared memory that Vetter contributed in his student research project [82] (see Section 5.2.4).

Having build the set I, we compute the set E_{new} of shortcuts that are necessary for the nodes in I (see Line 11). Having done that, we mark the incident edges of the nodes in I appropriately as upward and downward edges, remove all nodes in I and their incident edges from R, and insert or merge all necessary shortcuts respectively (see Lines 13 to 16). Before we begin with the next iteration by choosing the next independent node set I in the changed remaining graph R, we update the tentative costs of the neighbors of the just contracted nodes. To do so, we have to simulate the contraction of these neighbors of course (see Line 17).

During the node ordering, we successively set up the relation $\prec$ (see Line 10). Let $I_1, \dots, I_k$ be the sequence of the repeatedly chosen independent node sets ordered by time of creation. For $u \in I_i$ and $v \in I_j$ with $i < j$ the node order relation must fulfill $u \prec v$. For $u, v \in I_i$ we can choose freely between $u \prec v$ and $v \prec u$. Representing $\prec$ as a sequence of nodes and simply appending the nodes of the current set I to this sequence yields a node ordering that conforms with these thoughts.

Maintaining the Tentative Costs. The contraction of a node x changes the edges of its adjacent nodes in R, and (as said above) we have to update the tentative costs of these adjacent nodes. More precisely, the contraction of x means that every node $u \in N_R^1(x)$ loses at least an edge $u \to x$ or $x \to u$ and may gain one or more shortcuts. So, we have to update the tentative cost *u.cost* of all nodes $u \in N_R^1(x)$ because *u.cost* estimates how the contraction of u would change R (and this depends on the edges that u has in R). But updating this estimate includes a simulated contraction of u. Having finished the simulated contraction of u, we compute the new tentative cost of u as a linear combination of four *cost terms*.

Mainly, the four cost terms are chosen in a way that we obtain a hierarchy which is flat and sparse (as mentioned earlier in this thesis). As a consequence, the paths from the bottom of the hierarchy to its top are not too long and the hierarchy does not contain too many edges. Such hierarchies hasten our query algorithms, as these algorithms are based on bidirectional upward search from start and destination node. However, to achieve a lower memory usage, a low complexity of TTFs is also important. The four cost terms *edge quotient*, *hierarchy depth*, *unpacked edge quotient*, and *complexity quotient*—here, sorted by importance—try to ensure these properties and are explained in the following.

These four cost terms replace the cost terms of the original CHs [43] and have been introduced by Vetter when he parallelized the TCH preprocessing during a student research project [82]. The hope is that they suit time-dependent road networks better than the original terms do.

Edge Quotient. This term helps to keep the hierarchy sparse. When a node u were contracted, all its incident edges would be removed from R and then some shortcuts would be inserted. Accordingly, the *edge quotient*

$$\mathit{Edges}(u) \;:=\; \frac{\mathit{inserted}_R(u)}{\max\left\{1, \mathit{removed}_R(u)\right\}} \tag{5.12}$$

with

$$\begin{aligned}\mathit{inserted}_R(u) &:= |E_u| \\ \mathit{removed}_R(u) &:= |\{x \to u, u \to x \in E_R\}| ,\end{aligned}$$

where E_u is as defined by Equation (5.2), expresses how the number of edges would be changed locally by the contraction of u. Note that the edge quotient works better than the more intuitive term *edge difference*

$$\mathit{inserted}_R(u) - \mathit{removed}_R(u) \tag{5.13}$$

would do. This is because the values of the difference could get so large that other terms would not have enough influence any more.

Hierarchy Depth. Only considering the removed and inserted edges, one may get quite slow queries, as the resulting hierarchy might be sparse but not flat. So, we preferably contract nodes that are not so far away from the bottom of the hierarchy to prevent long upward or downward paths (with "long" in terms of hops). To ensure this, we maintain an attribute $u.\mathit{depth}$ of every node u. At the very beginning of the node ordering we set $u.\mathit{depth} := 1$ for all nodes u. Whenever a node x is really contracted we update $u.\mathit{depth}$ of all its neighbors $u \in N_R^1(x)$ by

$$u.\mathit{depth} \;:=\; \max\{x.\mathit{depth} + 1, u.\mathit{depth}\} . \tag{5.14}$$

The cost term *hierarchy depth* is simply the current value $\mathit{Depth}(u) := u.\mathit{depth}$. Obviously, $u.\mathit{depth}$ can be maintained *without* simulated contraction.

Note that the way we repeatedly choose the independent set I during node ordering, helps to distribute the node contractions uniformly over the remaining graph. This should already keep the resulting hierarchy quite flat.

Original Edge Quotient. When a node u is contracted, all the inserted and maybe some of the removed edges are shortcuts. The *original edge quotient* is the same as the edge quotient, but with the additional idea that all shortcuts count for the number of edges of the original path they represent; that is, for the number of edges that we get if we expand them completely. So, the idea is to compute something like

$$\frac{\sum_{\text{inserted shortcuts}} \#\text{original edges when expanded}}{\max\{1, \sum_{\text{edges removed from } R} \#\text{original edges when expanded}\}},$$

with the goal to balance the original path length of the shortcuts incident to a node. This should also support a uniform distribution of node contractions, which supports a flat hierarchy.

We maintain a value $orig[u \to v] \in \mathbb{N}$ for each edge $u \to v$ that we initially set to 1. Whenever a node x is really contracted, we set

$$orig[u \to v] := orig[u \to x] + orig[x \to v]$$

for each $u \to_{g*f} v \in E_x$, provided that no edge $u \to_h v$ is already present. Otherwise, we leave $orig[u \to v]$ unchanged.[5] The original edge quotient can then be computed by

$$Orig(u) := \frac{\sum\limits_{v \to w \in E_u} orig[v \to u] + orig[u \to w]}{\max\left\{1, \sum\limits_{v \to u \in E_R} orig[v \to u] + \sum\limits_{u \to w \in E_R} orig[u \to w]\right\}} \tag{5.15}$$

after a simulated contraction of u is performed, which means that the set E_u has been computed.

Complexity Quotient. This is the last of the four terms. To keep the complexity of the TTFs in a TCH low, we use the *complexity quotient*

$$Complex(u) := \frac{\sum\limits_{v \to_h w \in E_u} |h|}{\max\left\{1, \sum\limits_{v \to_f u \in E_R} |f| + \sum\limits_{u \to_g w \in E_R} |g|\right\}} \tag{5.16}$$

to measure how the number of TTF bend points in the TCH structure would change locally when the node u would be contracted. The goal is to save memory,

[5]Instead, one could set $orig[u \to v] := \max\{orig[u \to v], orig[u \to x] + orig[x \to v]\}$, for example. But whether this yields TCHs of better quality (i.e., less space and/or smaller query times) is not known.

because the hierarchy needs less space the fewer bend points the present TTFs have.

Note that this may also speed up preprocessing because the TTP searches being performed as witness searches during preprocessing (see Section 5.2.2) get slower the more bend points the processed TTFs have. This is discussed in more detail in the context of the running time of approximate TTP search (see Section 4.2.4). As a positive side effect, a low complexity of TTFs in the TCH may also speed up the answering of one-to-one TTP queries *after* preprocessing (see Section 5.3.2). The experiments performed for this thesis, however, do not consider whether witness searches and TTP queries really get faster.

Combining the Cost Terms. As said before, we use a linear combination of the above four cost terms to compute the tentative cost of a node. Of course, different configurations are possible for assigning values to the coefficients. We use the configuration

$$2\cdot \mathit{Edges}(u) \;+\; \mathit{Depth}(u) \;+\; \mathit{Orig}(u) \;+\; 2\cdot \mathit{Complex}(u) \tag{5.17}$$

that Vetter found by trial and error. We have not performed a systematic exploration.

Caching Results of Witness Searches. Node ordering takes much more time than TCH construction, where the node order $\prec$ is already known. This is mainly due to the simulated contractions that have to be performed for every node u whose neighbor x is really contracted. If the node u has multiple neighbors, then its contraction may be simulated multiple times—namely, if multiple neighbors of u are contracted before u. This, however, includes a lot of redundant work.

To understand that, consider a path $\langle v \to_f u \to_g w\rangle \subseteq R$. If multiple neighbors $x_1,\dots,x_k \neq v,w$ of u are really contracted, then k simulated contractions of u are performed. All these simulated contractions perform a witness search to check, whether a shortcut would be necessary for $\langle v \to_f u \to_g w\rangle$ or not. If neither $v \to u$ nor $u \to w$ are ever merged with a necessary shortcut edge, then the result of this check always stays the same, because EA times never change in the remaining graph when a node is contracted.

To avoid repeated witness searches for paths $\langle v \to_f u \to_g w\rangle$, we *cache* the results of witness searches by maintaining a value

$$\mathit{isShortcutNecessary}\big[(v,u,w)\big] \in \{\mathit{true}, \mathit{false}, \perp\}\ .$$

It tells us whether a shortcut $v \to w$ is necessary when u is contracted. The symbol $\perp$, which is the initial value, means that no witness search has been performed for the path $\langle v \to_f u \to_g w\rangle$ yet. When a node u is contracted—regardless whether it is a simulated or a real contraction—one first checks the cache for each $\langle v \rightarrow_f u \rightarrow_g w\rangle \subseteq R$ before an expensive witness search is performed. Of course, the cached result must be invalidated if $v \rightarrow_f u$ or $u \rightarrow_g w$ is merged with a necessary shortcut. More precisely, we set *isShortcutNecessary*$[(v,u,w)] := \bot$ in this case. The cached result is also invalidated if one of the nodes v,u,w is really contracted. Depending on how *isShortcutNecessary* is implemented (for example, using a hash table[6]), this can save memory.

It must be noted that calculating the complexity quotient (see Equation (5.16)) after the simulated contraction of u requires that $|g * f|$ is known for all $u \rightarrow_{g*f} v \in E_u$. If accessing *isShortcutNecessary*$[(v,u,w)]$ yields *true*, then $g * f$ has to be recalculated to obtain $|g * f|$. To avoid that $g * f$ is calculated again and again, we rather maintain a value

$$\mathit{complexity}\big[(v,u,w)\big] \in \mathbb{N}_0 \cup \{\bot\} ,$$

which can be deleted (or, equivalently, set to $\bot$) after one of the nodes u,v,w is really contracted. The initial value is $\bot$. If $g * f$ is computed for the first time, we set

$$\mathit{complexity}\big[(v,u,w)\big] := |g * f| .$$

If $v \rightarrow_f u$ or $u \rightarrow_g w$ is merged with a newly created shortcut, *complexity*$[(v,u,w)]$ must be invalidated; that is, set to $\bot$.

Caching the results of witness searches by maintaining *isShortcutNecessary*$[\cdot]$ and *complexity*$[\cdot]$ has been introduced by Vetter when he was working on the preprocessing as a student assistant. Our experiments show that this caching greatly reduces the running time of the preprocessing (see Section 5.6).

Reusing a Node Order. We already said that the node ordering not only yields the node order relation $\prec$ but also a complete TCH structure and that it is not necessary to perform a separate construction process afterwards. However, if we have different sets of TTFs for the road network G, we can compute a node order for only one of these sets. Then, we can *reuse* this order for all the other sets of TTFs and perform the construction process without a further node ordering each. This is likely to increase the running times needed for construction and also for querying, as the resulting TCH may have a somewhat worse quality. But it may work well enough and our experiments confirm that this can be the case in practice (Section 5.6). This way, a considerable amount of running time can be saved during preprocessing.

[6]A *hash table* is a well-known fundamental data structure allowing random access in expected constant time (e.g., see Mehlhorn and Sanders [61]).

5.2.4 Parallel Preprocessing in Shared Memory

TCH construction (see Section 5.2.2) and node ordering (see Section 5.2.3) as described in this thesis are very well suited for parallel execution in shared memory. This is enabled by iteratively choosing independent sets I as characterized by Equation (5.10) or (5.11). This makes sure that the contraction of a node $x \in I$ does not influence the contraction of any other node in I. The details are explained in the following. Note that this parallelization of TCH construction and node ordering, including how the set I is chosen, has been contributed by Vetter during a student research project [82].

Parallel TCH Construction. In case of TCH construction (where $\prec$ is already known) the set I is build as in Equation (5.10). Contracting $x \in I$ has no influence on the contraction of any other node in I because of two reasons. First, I is an independent set in R; that is, no two nodes in $x, y \in I$ are adjacent in R. So, no node $y \in I \setminus \{x\}$ loses or gains any edges when the node x is contracted. Second, the contraction of x does not alter the EA times in the remaining graph R (see Lemma 5.1). This means that the witness searches performed for any other node $y \in I \setminus \{x\}$ yield the same results regardless whether y is contracted before or after x. The question whether a shortcut $u \rightarrow_{g*f} v$ is necessary to replace a path $\langle u \rightarrow_f y \rightarrow_g v \rangle \subseteq R$ is hence independent from the contraction of x. So, all nodes in I can be contracted in an arbitrary order or even *in parallel* without altering the result of this process.

With respect to the above argument, it is obvious that set E_x as defined by Equation (5.2) can be computed in parallel for all $x \in I$. The affected parts of Algorithm 5.1, which shows pseudocode for TCH construction, are Lines 8, 9, and 11. Note that the access to the remaining graph R is read-only in this case. The set I can also be computed in parallel, because the access to the node order relation $\prec$ is read-only, too (see Line 6). How far the updating of R and H (see Line 10 and 12 to 21) can be performed in parallel, depends on the graph data structures used to represent R and H.

Parallel Node Ordering. Like in case of TCH construction, all nodes in I can be contracted in parallel, because Equation (5.11), just like Equation (5.10), ensures that I is an independent set. In case of parallel node ordering, however, it can happen that two threads simulate the contraction of the same node—at least, if the computation is assigned to the threads by assigning each node $x \in I$ and simulating the contraction of all nodes $u \in N_R^1(x)$ with the same thread. To prevent such redundant simulated contractions, Equation (5.11) ensures a 3-hop gap between the nodes in an I. Another solution would be to assign each node $u \in \bigcup_{x \in I} N_R^1(x)$

to a thread directly. In this case, however, one would have to check for duplicate nodes, which our solution avoids.

The parts of node ordering (see Algorithm 5.3) that can be parallelized easily, are the computation of the set I (see Line 9), the computation of the set E_{new} (see Line 11), and the simulated contractions (see Line 17). Whether R and H can, at least partly, be updated in parallel, depends on the graph data structure used to represent R and H—just like in case of TCH construction.

5.2.5 Differences to CHs with Constant Travel Costs

We complete the discussion of TCH preprocessing with a comparison to the preprocessing of the original CHs [43, 44], which is designed to deal with constant travel costs. There are four main issues where TCH preprocessing is different from original CH preprocessing.

Node Ordering without PQ. Just like TCHs, original CHs use tentative node costs ("priorities") to determine which nodes are most attractive to be contracted next during node ordering. However, original CHs do not build an independent node set before contraction. Instead, all nodes not yet contracted are stored in a PQ S using the tentative node cost as key. The node x to be contracted next is, in principle, obtained from the PQ by invoking $x := S.\mathit{deleteMin}()$. On the one hand, using a PQ to govern the node ordering has the advantage that the next node to be contracted can be obtained easily. On the other hand, the PQ does not support the selection of nodes that can be contracted independently. In other words, a node ordering process governed by a PQ is not well suited for parallel node contraction. This is why Vetter dropped the PQ when parallelizing the TCH preprocessing in his student research project [82].

Never Updating Tentative Costs Lazily. Original CHs allow cost terms that not only depend on the local neighborhood of a node but on a wider area in the remaining graph R during preprocessing. This means, it is not necessarily clear when a tentative node cost changes. So, tentative node costs cannot always be kept up-to-date efficiently. Instead, tentative node costs are updated "lazily": Whenever original CHs select a node x with minimal $x.cost$ for contraction, it is not contracted immediately. Instead, the tentative node cost $x.cost$ is recalculated first. If the tentative node cost increases such that $x.cost$ is no longer minimal amongst all nodes not contracted so far, then another node x with minimal $x.cost$ is selected. This process is repeated until $x.cost$ stays minimal for a node x, and this node is then contracted. If a PQ S is used to govern the node ordering, this process works as follows: A node x with minimal $x.cost$ is obtained by invoking

$x := S.deleteMin()$. If $x.cost > S.min()$ gets true after recalculating $x.cost$, then the node x is reinserted into the PQ by performing $S.insert(x, x.cost)$. This process is iterated until the minimum of the PQ gets stable and the then minimal node is contracted.

If tentative node costs can only increase during preprocessing, then lazy updates are enough to preserve the resulting node order; that is, updating tentative node costs at once, which may not be very efficient, would yield the same node order. If tentative node costs can also decrease, however, then lazy updates may alter the resulting node order. To keep the difference low, original CHs perform a complete update of all tentative node costs from time to time. In case of TCHs, things are simpler. This is because all cost terms used by TCH preprocessing only depend on the direct neighborhood of a node. It is hence enough that TCH preprocessing only updates the cost of a node when one or more of its neighbors get contracted. Lazy updates or occasional complete updates are not necessary.

Different Cost Terms. The only cost term of TCHs that is adopted from original CHs without any substantial modification is the hierarchy depth (see Equation (5.14)), which is called "cost of queries" in the context of original CHs. The edge quotient (see Equation (5.12)) is also adopted from original CHs, but with the change that original CHs calculate a difference instead of a quotient (see Equation (5.13)). The original edge quotient (see Equation (5.15)) combines the edge quotient with an idea also adopted from original CHs; namely, the idea to take the length of paths represented by shortcuts into account.

The complexity quotient (see Equation (5.16)) has no direct counterpart in the context of original CHs. The idea behind this term is to reduce the time spend on witness search, whose running time increases with number of present bend points. Original CHs also consider the time spend on witness search, but in terms of sizes of search spaces that occur during witness searches. Though considering the search space size would also be likely to reduce the running times of TCH preprocessing, it not only depends on the direct neighborhood of a node to be contracted but on a wider area. In the context of TCHs, however, all cost terms only depend on the direct neighborhood of a node, as explained above. So, considering search space sizes during TCH preprocessing would destroy this convenient property, and keeping the tentative node costs up-to-date would be much more difficult hence. Another goal of the complexity quotient is to reduce the memory usage of TTFs in the resulting hierarchy. This is not an issue in the context of original CHs, of course, because they only deal with constant travel cost.

Just like in case of TCHs, the node ordering of original CHs tries to create hierarchies that are not only sparse but flat. In case of TCHs, this goal is greatly supported by the iterated selection of independent sets of nodes to be contracted, which should help to distribute the node contractions uniformly over the graph. In case of original CHs, however, the selection of the next node to be contracted is governed by a PQ that is not necessarily aware of uniformly distributed contractions. To overcome this, original CHs use cost terms that support uniformity. These cost terms, which can be be quite complicated, are not adopted by TCHs. Nevertheless, one cost terms used by TCHs—namely, the original edge quotient—should also encourage uniformity.

Different Optimizations of Witness Search. Witness search is the bottleneck of CH preprocessing. This applies both to the constant case (i.e., to original CHs [43, 44]) and to the time-dependent case (i.e., to TCHs). However, original CHs and TCHs utilize quite different techniques to make the witness search faster. Consider the contraction of a node x. In case of original CHs, witness search is performed as a "one-to-several" query. That is, for each node u with $u \to x \in E_R$, a single Dijkstra search is performed to compute the shortest path distances $\mu_R(u,v)$ for all $v \in T_x := \{w \,|\, x \to w \in E_R\}$. To make this faster, original CHs perform a "simple, asymmetric form of bidirectional search" [43] that combines a single-hop backward search with a bucket structure adopted from hierarchical many-to-many querying [43, 44, 56, 78] (see Section 1.3.1 on page 31 for a short summary). Note that original CHs apply a "staged" hop limit to the witness search; that is, the hop limit may change during preprocessing.

In case of TCHs, witness search performs one-to-one queries. On the one hand, this has the drawback that a separate computation must be performed for all paths $\langle u \to x \to v \rangle \subseteq R$. On the other hand, this has the advantage that heuristic thinning of corridors gets applicable (see Section 5.2.2 on page 195). Another optimization utilized by TCHs, but not by original CHs, is that the results of witness searches are cached (see Section 5.2.3 on page 202). Heuristic thinning of corridors and caching the results of witness searches, which have both been provided by Vetter, make the slow TTP query much faster during witness search. This is confirmed by our experiments (see Section 5.6). Note that the hop limit applied to TCH preprocessing (see Section 5.2.2 on page 196) is not staged; that is, the hop limit does not change over time while preprocessing runs.

5.3 Exact Querying

In this section we explain how TCH structures can be used to perform fast and exact EA and TTP queries. We achieve this by using bidirectional upward search, which is a bidirectional search where we only relax edges that lead upward in the hierarchy. As said before, this is fast if the TCH structure H provided by the preprocessing is flat and sparse. The guaranteed existence of prefix-optimal

EA *up-down-paths* in TCH structures (see Theorem 5.5) ensures that the query algorithms described in this section compute correct results. EA up-down-paths are analogous to shortest up-down-paths, which we mention earlier in the context of constant travel cost CHs (see Section 2.3.2).

For EA queries, bidirectional search has the problem that exact time-dependent backward search would require us to know the arrival time, which is part of what we want to compute. To overcome this, we apply approximation. More precisely, we use time-dependent Dijkstra only as forward search, but TTP interval search as backward search because it does not require a given arrival time. Even though this does not yield an EA path yet, we know that the predecessor graphs of forward and backward search together contain an EA up-down-path for the given departure time. All what is left to do is to perform a further time-dependent Dijkstra search on the edges touched by the backward search (see Section 5.3.1).

TCH-based TTP queries are conceptually simpler than EA queries, though more expensive to compute. The simplicity is due to the fact that bidirectional upward TTP search works fine, because TTP search does not require a fixed arrival time if used as backward search. As a result, TTP queries on TCHs are straightforward. However, we can do better: Before we perform the bidirectional TTP search, we perform a bidirectional TTP interval search first. This yields two relatively small cones in that we perform the bidirectional TTP search. This way, the TTP query needs much less time, because much less edges have to be processed by the bidirectional TTP search. This is an application of the idea that fast approximate computations speed up slow exact ones (see Section 5.3.2).

Both EA and TTP queries can be made faster by applying a simple but effective pruning technique called *stall-on-demand*. There, we try to detect nodes that can not lie on an EA up-down-path from the start to the destination node with prefix-optimal upward part (see Section 5.3.3).

5.3.1 Earliest Arrival Queries

Given a start node s, a destination node t, and a departure time τ_0 we want to compute $\mathrm{EA}_G(s,t,\tau_0)$ as well as a corresponding EA path in G. Using TCHs we can do this very fast as follows: First, we run Algorithm 5.4 in H to obtain an (s,t,τ_0)-EA up-down-path. However, an EA up-down-path usually contains artificial shortcut edges and is not a valid route with respect to the original road network G. The EA up-down-path has to be expanded hence. To do so, we run Algorithm 5.7 that recursively expands the EA up-down-path for departure time τ_0. Both steps take very little time.

Computing an EA Up-Down-Path. The computation of an (s,t,τ_0)-EA up-down-path (see Algorithm 5.4) is the main task of answering EA queries. It runs

Algorithm 5.4. Computes an (s,t,τ_0)-EA up-down-path in $H = H_\uparrow \cup H_\downarrow$. As subroutines the procedures *tdRelax* (see Algorithm 5.5) and *ttpIntervalRelax* are invoked (see Algorithm 5.6).

```
1  function tchEaQuery($s,t : V$, $\tau_0 : \mathbb{R}$) : Path
2      $\tau[u] := \infty$ for all $u \in V$, $\tau[s] := \tau_0$
3      $[q[u], r[u]] := [\infty, \infty]$ for all $u \in V$, $[q[t], r[t]] := [0,0]$
4      $p_s[u] := \bot$, $p_t[u] := \emptyset$ for all $u \in V$
5      $B := \infty$                                   // upper bound of EA time
6      $X := \emptyset : Set$                           // set of candidate nodes
7      function downwardSearch() : Path
8          $\tau_{\text{down}}[u] := \infty$ for all $u \in V$
9          $Q_{\text{down}} := \emptyset : PriorityQueue$          // PQ for downward search
10         foreach $u \in X$ do
11             if $\tau[u] + q[u] \le B$ then
12                 $\tau_{\text{down}}[u] := \tau[u]$, $Q_{\text{down}}.insert(u, \tau[u])$
13         while $Q_{down} \neq \emptyset$ do
14             $u := Q_{\text{down}}.deleteMin()$
15             if $u = t$ then return $\langle s = p_s[\dots p_s[t] \dots] \to \dots \to p_s[t] \to t \rangle$
16             for $v \in p_t[u]$ do $tdRelax(u \to_f v, \tau_{\text{down}}, p_s, Q_{\text{down}})$
17     $Q_s := \{(s, \tau_0)\}$, $Q_t := \{(t, 0)\} : PriorityQueue$
18     $\Delta := t$                                    // current search direction
19     while $(Q_s \neq \emptyset$ or $Q_t \neq \emptyset)$ and $\min\{Q_s.min(), Q_t.min()\} \le B$ do
20         if $Q_{\neg\Delta} \neq \emptyset$ then $\Delta := \neg\Delta$    // change of direction: $\neg s := t$ and $\neg t := s$
21         $u := Q_\Delta.deleteMin()$
22         if $B < \infty$ and $\tau[u] + q[u] \le B$ then $X := X \cup \{u\}$
23         $B := \min\{B, \tau[u] + r[u]\}$
24         for $u \to_f v \in E_\Delta$ do              // with $E_s := E_\uparrow$ and $E_t := E_\downarrow^\top$
25             if $\Delta = s$ then $tdRelax(u \to_f v, \tau, p_s, Q_s)$
26             else $ttpIntervalRelax(u \to_f v, q, r, p_t, Q_t)$
27     if $B = \infty$ then return $\langle\rangle$      // Did forward and backward search meet?
28     return downwardSearch()
```

in two phases. The first phase (see Lines 17 to 26) is a time-dependent bidirectional upward search in the TCH structure H. The second phase (see Lines 7 to 16 and Line 28) is a time-dependent Dijkstra search that only relaxes edges touched by the backward search of the first phase. Note that the downward search is encapsulated in a nested procedure to improve the readability of the pseudocode.

Algorithm 5.5. Edge relaxation procedure as in time-dependent Dijkstra search (see Algorithm 4.1). The *Reference* parameters τ, p, Q are necessary to provide context information of the calling time-dependent Dijkstra search. These are references to the respective structures storing label and predecessor information, as well as to the respective PQ.

```
procedure $tdRelax(u \rightarrow_f v : Edge,\ \tau, p, Q : Reference)$
    if $\mathbf{arr}\, f(\tau[u]) \geq \tau[v]$ then continue
    $\tau[v] := \min\{\tau[v],\ \mathbf{arr}\, f(\tau[u])\}$
    $p[v] := u$
    if $v \notin Q$ then $Q.insert(v,\ \tau[v])$
    else $Q.updateKey(v,\ \tau[v])$
```

Algorithm 5.6. Edge relaxation procedure as in TTP interval search (see Algorithm 4.3). The *Reference* parameters q, r, p, Q are necessary to provide context information of the calling TTP interval search. This is similar to Algorithm 5.5 with the difference that the label information of a node u consists of intervals $[q[u], r[u]]$ here.

```
procedure $ttpIntervalRelax(u \rightarrow_f v : Edge,\ q, r, p, Q : Reference)$
    $[q_{\text{new}}, r_{\text{new}}] := [q[u] + \min f,\ r[u] + \max f]$
    if $q_{\text{new}} > r[v]$ then return
    if $r_{\text{new}} < q[v]$ then $p[v] := \emptyset$
    $p[v] := \{u\} \cup p[v]$
    if $q_{\text{new}} \geq q[v]$ and $r_{\text{new}} \geq r[v]$ then return
    $[q[v], r[v]] := \big[\min\{q[v], q_{\text{new}}\},\ \min\{r[v], r_{\text{new}}\}\big]$
    if $v \notin Q$ then $Q.insert(v,\ q[v])$
    else $Q.updateKey(v,\ q[v])$
```

Phase 1: Bidirectional Upward Search. The forward search is a time-dependent Dijkstra (see Section 4.2.1) starting from s running in $H_\uparrow$. The backward search is a backward TTP interval search starting from t running in $H_\downarrow$, which means that it actually runs in $H_\downarrow^\top$ (see Section 4.3.2). In contrast to the forward search, the backward search yields only approximate results. This is necessary because we do not know the EA time at t, which is part of what we want to compute. Both searches only relax edges that lead upward in the TCH. In case of the backward search, all edges in $H_\downarrow$ are relaxed in reverse direction of course. Note that forward and backward search are performed alternately. This is controlled by the variable $\Delta \in \{s,t\}$ that encodes the current direction in terms of the start node of the respective search (s for forward and t for backward search).

To prevent rather longish pseudocode, the edge relaxation of forward and backward search has been factored out (see Lines 25 and 26). The respective pseudocode can each be found in Algorithm 5.5 and 5.6, which are both invoked by Algorithm 5.4 as subprocedures. Note that Algorithm 5.5 and 5.6 need access to context information from Algorithm 5.4 to work correctly. That is, access to

- the respective node label information,
- the respective predecessor information, and
- the respective PQ.

This is reflected by the fact that signatures of Algorithm 5.5 and 5.6 contain some *Reference* parameters allowing to pass the respective data structures. Note that passing *Reference* parameters does only cause constant overhead, because one does not pass the structures themselves but references to the structures.

The nodes where forward and backward search meet are called *candidate nodes*. More precisely, a candidate node u is a node with $\tau[u] + r[u] < \infty$. We store the candidate nodes in the *candidate set* X (see Line 22). During the bidirectional search we maintain an upper bound $B \geq \mathrm{EA}_G(s,t,\tau_0)$ initialized with ∞. If a node u is settled by forward or backward search (see Line 23), we update B by setting

$$B := \min\{B, \tau[u] + r[u]\} \ .$$

The bidirectional search can be stopped as soon as the minima of forward and backward PQ both exceed B, because they can no more contribute to a better up-down-path then (see Line 19). The upper bound B can be used to rule out candidate nodes that can not lie on an (s,t,τ_0)-EA up-down-path (see condition in Line 22). If $B = \infty$ holds after finishing the while loop, then no up-down-path from s to t exists in H (see Line 27). Otherwise, we can be sure that $H_{\uparrow}(p_s) \cup (H_{\downarrow}^{\top}(p_t))^{\top}$ (i.e., the predecessor graph of the forward search together with the transpose predecessor graph of the backward search) contains at least one (s,t,τ_0)-EA up-down-path. Finding such a path is the task of the second phase.

Phase 2: Downward Search. The second phase, or *downward search* (see Lines 7 to 16), is actually a time-dependent Dijkstra search in the transpose predecessor graph of the backward search; that is, in $(H_{\downarrow}^{\top}(p_t))^{\top} \subseteq H_{\downarrow}$. Note that $(H_{\downarrow}^{\top}(p_t))^{\top}$ needs not to be build before we run the downward search.[7] Instead, the predecessor information p_t can be used directly. This is reflected in Line 16. The downward search has multiple start nodes; namely, the candidate nodes, which are inserted in the PQ Q at Lines 10 to 12. As soon as the downward search takes t out of the PQ, an (s,t,τ_0)-EA up-down-path has been found (see Lines 15). Figure 5.5 illustrates how the two phases of the TCH-based EA query work.

[7]Recall that we do this during preprocessing, where we use BFS to build a corridor for the witness search (see Algorithm 5.2 in Section 5.2.2).

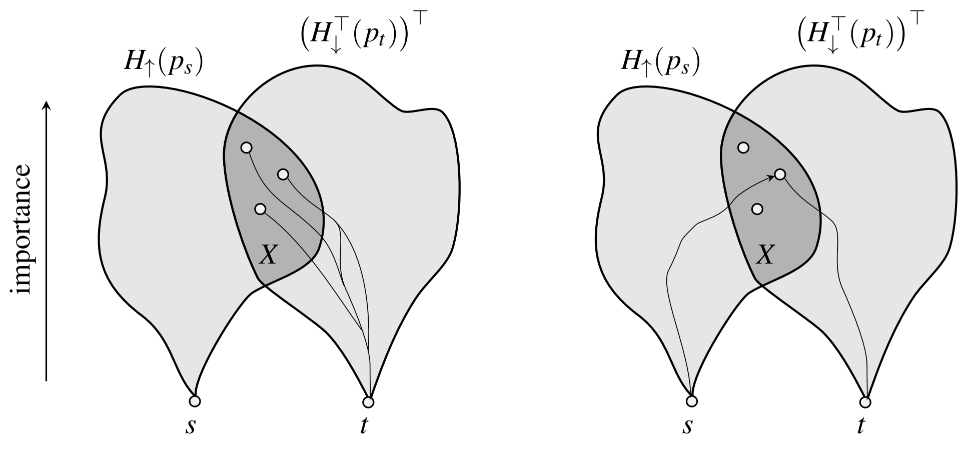

$H_\uparrow(p_s)$
$(H_\downarrow^\top(p_t))^\top$
importance
X
s
t
$H_\uparrow(p_s)$
$(H_\downarrow^\top(p_t))^\top$
X
s
t

Figure 5.5. How the TCH-based EA query (see Algorithm 5.4) works. We look at the TCH structure H “from the side” with the more important nodes higher up. Predecessor graphs are depicted in gray. Left: The set of candidate nodes X contains the nodes where forward and backward search meet; that is, the nodes that lie in the intersection of the predecessor graph $H_\uparrow(p_s)$ and the transpose predecessor graph $(H_\downarrow^\top(p_t))^\top \subseteq H_\downarrow$. The downward search only searches along paths in $(H_\downarrow^\top(p_t))^\top$ starting from nodes in X. Right: Having finished the downward search, an (s,t,τ_0)-EA up-down-path is found. Its top node is one of the candidate nodes in X and it is completely contained in $H_\uparrow(p_s) \cup (H_\downarrow^\top(p_t))^\top$.

Note that we again use the upper bound B to rule out candidate nodes there. This may remove the one or another candidate node $x \in X$, because B may have been lowered since x has been inserted in X. It is interesting to note that the downward search does not maintain its own predecessor information. Instead it uses the predecessor information p_s of the upward search.

Correctness. With the guaranteed existence of a prefix-optimal (s,t,τ_0)-EA up-down-path in H, which is ensured by the preprocessing (see Section 5.2.2), we can prove that Algorithm 5.4 really computes an EA up-down-path.

Theorem 5.6. *Let H be a TCH, $s,t \in V$ such that t is reachable from s, and $\tau_0 \in \mathbb{R}$. Then, Algorithm 5.4 returns an (s,t,τ_0)-EA up-down-path in H.*

Proof. According to Theorem 5.5 there exists a prefix-optimal (s,t,τ_0)-EA up-down-path with top node x_0 in H. So, x_0 is reached both by forward and backward search. Thus, we have $x_0 \in X$ and $\tau[x_0] = \mathrm{EA}_H(s,x_0,\tau_0)$. Due to Lemma 4.36 we also know $(H_\downarrow^\top(p_t))^\top \subseteq H_\downarrow$ contains a path $P_0 = \langle x_0 \to \cdots \to t\rangle$ that is an $(x_0,t,\mathrm{EA}_H(s,x_0,\tau_0))$-EA-path not only in $H_\downarrow$ but also in H. The latter is because P_0 is suffix of an (s,t,τ_0)-EA-path in H. Now, consider the graph H_0 formed by $(H_\downarrow^\top(p_t))^\top$, the start node s, and by an edge $s \to_{f_x} x$ with $f_x :\equiv \tau[x] - \tau_0 =$

$\mathrm{TT}_{H_\uparrow}(s,x,\tau_0)$ for each $x \in X$; especially, it is $f_{x_0} \equiv \tau[x_0] - \tau_0 = \mathrm{TT}_H(s,t,\tau_0)$. We consider the concatenated path $R_0 := \langle s \rightarrow_{f_{x_0}} x_0 \rangle P_0$ and calculate

$$\begin{aligned}\mathbf{arr}\, f_{R_0}(\tau_0) = \mathbf{arr}(f_{P_0} * f_{x_0})(\tau_0) &= \mathbf{arr}\, f_{P_0}\big(\mathrm{EA}_H(s,x_0,\tau_0)\big)\\ &= \mathrm{EA}_H\big(x_0,t,\mathrm{EA}_H(s,x_0,\tau_0)\big) = \mathrm{EA}_H(s,t,\tau_0)\ .\end{aligned}$$

Thus, it is $\mathrm{EA}_{H_0}(s,t,\tau_0) = \mathrm{EA}_H(s,t,\tau_0)$, because there are no better EA times in H_0 than in H for departure time τ_0 by construction. But it is easy to see that the downward search is equivalent to a time-dependent Dijkstra in H_0 starting from s. Thus, Algorithm 5.4 terminates with $\tau_{\mathrm{down}}[t] = \mathrm{EA}_H(s,t,\tau_0)$ returning an up-down-path $P_\uparrow P_\downarrow$ with $P_\uparrow := \langle s = p_s[\dots p_s[x]\dots] \rightarrow \cdots \rightarrow p_s[x] \rightarrow x\rangle \subseteq H_\uparrow$ and $P_\downarrow := \langle x = p_s[\dots p_s[t]\dots] \rightarrow \cdots \rightarrow p_s[t] \rightarrow t\rangle \subseteq H_\downarrow$ for some $x \in X$ fulfilling

$$\begin{aligned}\mathrm{EA}_H(s,t,\tau_0) = \tau_{\mathrm{down}}[t] &= \mathbf{arr}\, f_{P_\downarrow}(\tau[x])\\ &= \mathbf{arr}\, f_{P_\downarrow}\big(\mathbf{arr}\, f_{P_\uparrow}(\tau_0)\big) = \mathbf{arr}\, f_{P_\uparrow P_\downarrow}(\tau_0)\ ,\end{aligned}$$

because of Lemma 4.1. So, Algorithm 5.4 returns the desired result.

Note that the stopping condition of the while loop (Line 19) does not affect the correctness. This follows from the typical monotonous behavior of the Dijkstra-like algorithms considered in this thesis (see Lemma 4.5 and 4.35). Also, the ruling out of candidate nodes in Lines 10 to 12 and 22 only applies to nodes that can not lie on an (s,t,τ_0)-EA-path. □

Expanding the EA Up-Down-Path. Having computed an (s,t,τ_0)-EA up-down-path using Algorithm 5.4, we run Algorithm 5.7 to expand this up-down-path to obtain the corresponding original path $\langle s \rightarrow \cdots \rightarrow t\rangle \subseteq G$. The idea is to recursively replace all the artificial shortcut edges by the two-hop paths they represent. In principle, this is the same idea as expanding shortcuts for constant travel costs (see Section 2.3). In the time-dependent case, however, it is a little more complicated, because a represented two-hop path depends on the time one departs from the source node of a shortcut. To provide the necessary time-dependent information about middle nodes, the preprocessing annotates every edge with a shortcut descriptor (see Section 5.2.1 and 5.2.2 for a detailed explanation).

Shortcuts must be expanded differently for different departure times. For this reason, we maintain the departure time for the currently processed shortcut edge during expansion. To do so we use the variable τ_{current}. It is initialized with τ_0, which is the departure time at the start node s. Algorithm 5.7 realizes the recursion in an iterative manner using a stack. This means, we always process the left-most edge that is not fully expanded yet. As a result, τ_{current} needs only to be updated when an edge is fully expanded and then appended to the resulting original path (see Lines 11 to 14).

Algorithm 5.7. Expands all time-dependent shortcuts of a given path $\langle u_1 \rightarrow_{f_1} u_2 \rightarrow_{f_2} \cdots \rightarrow_{f_{k-1}} u_k \rangle$ in H. The result is the corresponding original path in the original road network G. The result of the expansion depends on the given departure time τ_0. For different departure times, the shortcut may represent different paths, which may result in different original paths.

```
function expandPath($\langle u_1 \rightarrow_{f_1} u_2 \rightarrow_{f_2} \cdots \rightarrow_{f_{k-1}} u_k \rangle$ : Path, $\tau_0 : \mathbb{R}$) : Path
    result := $\langle\rangle$ : Path
    $\tau_{\text{current}} := \tau_0 : \mathbb{R}$
    for $i = 1, \ldots, k-1$ do
        $S := \emptyset$ : Stack                                    // a LIFO queue
        S.push($u_i \rightarrow_{f_i} u_{i+1}$)
        while $S \neq \emptyset$ do
            $u \rightarrow_f v$ := S.pop()
            let $D$ be the shortcut descripor of $u \rightarrow_f v$
            consider $x$ sucht that $D = \langle \ldots, ([a,b],x), \ldots \rangle$ with $a \leq \tau_{\text{current}} < b$
            if $x = \perp$ then
                append $u \rightarrow_f v$ to result
                $\tau_{\text{current}} := \tau_{\text{current}} + f(\tau_{\text{current}})$
                continue
            S.push($x \rightarrow_h v$) for $x \rightarrow_h v \in E_\uparrow$
            S.push($u \rightarrow_g x$) for $u \rightarrow_g x \in E_\downarrow$
    return result
```

Corollary 5.7. *Let H be a TCH constructed from a road network G, $P \subseteq H$ an up-down-path from s to t, and $\tau_0 \in \mathbb{R}$. Expanding P for departure time τ_0 recursively as specified in Algorithm 5.7 yields a path $P' \subseteq G$ from s to t with $f_{P'}(\tau_0) = f_P(\tau_0)$.*

5.3.2 Travel Time Profile Queries

Answering TTP queries with TCHs is more straightforward than in case of EA queries, but also more time-consuming. An arrival time is not required, which enables us to answer TTP queries by simply performing a bidirectional TTP search. A subsequent Dijkstra-like search, like in case of EA queries, is not needed. This yields a relatively simple basic procedure for answering TTP queries (see Algorithm 5.8). Though this is already feasible, we can still be much faster: To reduce the search space of the bidirectional TTP search, we perform a bidirectional TTP interval search first. The TTP search is still performed afterwards, but only on two considerably smaller cones that are extracted from the search spaces of the bidirectional TTP interval search (see Algorithm 5.10).

Basic TTP Querying. Given a start node s and a destination node t, we want to compute $\mathrm{TT}_G(s,t,\cdot)$. In theory, such TTP queries can be solved with the one-to-one version of TTP search (see Section 4.2.2). However, even for middle-sized road networks this is not feasible. Instead, we use TCHs to answer such queries (see Algorithm 5.8). This runs much faster as our experiments show (see Section 5.6). As in case of EA queries, we perform a bidirectional search on the TCH structure H, where we only go upward in the hierarchy. Here, both searches are TTP searches that run in $H_\uparrow$ and $H_\downarrow$, a forward TTP search (see Section 4.2.2) and a backward TTP search (see Section 4.3.1). Note that the pseudo code of for edge relaxations is, again, factored out (see Line 14 and 15 as well as Algorithm 5.9)

The labels of a node u with respect to forward and backward search are denoted by $F_s[u]$ and $F_t[u]$ each. A label $F_s[u]$ is the current tentative TTP for traveling from s to a node u. Analogously, $F_t[u]$ is the current tentative TTP for traveling from u to t. Of course, the backward search relaxes all edges in reverse direction, which means it runs on $H_\downarrow^\top$. Remember that the backward version of TTP search links TTFs the other way round than the forward version, because linking of TTFs is not commutative (see Section 4.3.1). This is reflected in Lines 14 and 15, where f is either linked from the left or from the right, depending on the

Algorithm 5.8. A TTP query using a TCH structure $H = H_\uparrow \cup H_\downarrow$. For a given start node s and a given destination node t this algorithm computes $\mathrm{TT}_G(s,t,\cdot)$. As subroutine the subprocedure *ttpRelax* (see Algorithm 5.9) is invoked.

```
1  function $tchTtpQuery(s,t : V) : TTF$
2      $F_s[u] := \infty$ for all $u \in V \setminus \{s\}$, $F_s[s] :\equiv 0$
3      $F_t[u] := \infty$ for all $u \in V \setminus \{t\}$, $F_t[t] :\equiv 0$
4      $B := \infty$                                        // upper bound of travel time
5      $X := \emptyset : Set$                                              // candidate set
6      $Q_s := \{(s,0)\}, Q_t := \{(t,0)\} : PriorityQueue$          // forw. and backw. PQ
7      $\Delta := t$                                                  // search direction
8      while $(Q_s \neq \emptyset$ or $Q_t \neq \emptyset)$ and $\min\{Q_s.min(), Q_t.min()\} \leq B$ do
9          if $Q_{\neg\Delta} \neq \emptyset$ then $\Delta := \neg\Delta$          // with $\neg s := t$ and $\neg t := s$
10         $u := Q_\Delta.deleteMin()$
11         if $B < \infty$ and $\min F_s[u] + \min F_t[u] \leq B$ then $X := X \cup \{u\}$
12         $B := \min\{B, \max F_s[u] + \max F_t[u]\}$
13         for $u \to_f v \in E_\Delta$ do                       // with $E_s := E_\uparrow$ and $E_t := E_\downarrow^\top$
14             if $\Delta = s$ then $ttpRelax(u, v, f * F_s[u], F_s, \perp, Q_s)$
15             else $ttpRelax(u, v, F_t[u] * f, F_t, \perp, Q_t)$            // ignore predecessors
16     compute $f_i := F_t[x_i] * F_s[x_i]$ for all $x_i \in \{x_1, \ldots, x_k\} := X$
17     return $\min(f_1, \min(f_2, \ldots \min(f_{k-1}, f_k) \ldots))$
```

Algorithm 5.9. Edge relaxation procedure as in TTP search (see Algorithm 4.2) and backward TTP search. Whether the relaxation works in the manner of forward or backward TTP search, is controlled by the parameter g_{new}. This is because linking of TTFs is not a commutative operation. For example, $f * F_s[u]$ is passed to g_{new} in case forward search, $F_t[u] * f$ in case backward search (as it happens in Algorithm 5.8). The *Reference* parameters F, p, Q are necessary to provide context information of the calling TTP search.

```
procedure $ttpRelax(u \to_f v : Edge, g_{\text{new}}, F, p, Q : Reference)$
    if $g_{\text{new}} \geq F[v](\tau)$ for all $\tau \in \mathbb{R}$ then return
    if $g_{\text{new}}(\tau) < F[v](\tau)$ for all $\tau \in \mathbb{R}$ then $p[v] := \emptyset$
    $p[v] := \{u\} \cup p[v]$            // remove or remember tentative predecessors of v
    $F[v] := \min(F[v], g_{\text{new}})$            // update tentative node label of v
    if $v \notin Q$ then $Q.insert(v, \min F[v])$
    else $Q.updateKey(v, \min F[v])$            // minimum of TTF is PQ key
```

search direction. It must also be noted that no predecessor information is maintained by Algorithm 5.8, because a TTP query only computes a TTF and no path. So, the respective *Reference* parameter p is not used in Line 14 and 15 and no predecessor information is passed to the relaxation procedure in Algorithm 5.9.

The minimum operation on TTFs is, in contrast to the linking operation, commutative. We exploit this to speed up Line 17: A PQ controls which two TTFs are combined using the minimum operation next. More precisely, we insert all f_i in the PQ using $|f_i|$ as key. Then, we repeatedly remove two TTFs f, g of minimal complexity from the PQ and insert the result $\min(f,g)$ with key $|\min(f,g)|$ until only one TTF is left. This is the sought-after result $\mathrm{TT}_H(s,t,\cdot)$. Note that this has similarities with a technique called *corridor contraction* described later in this chapter (see Section 5.4.3).

Theorem 5.8. *Let H be a TCH structure generated from G and $s,t \in V$. Then, the TCH-based TTP query described in Algorithm 5.8 returns* $\mathrm{TT}_G(s,t,\cdot)$.

Proof. From Theorem 5.5 we know that for all departure times $\tau \in \mathbb{R}$ there is a prefix-optimal (s,t,τ)-EA up-down-path in H with top node x_τ. So, x_τ is reached by forward and backward search with the final labels $F_s[x_\tau]$ and $F_t[x_\tau]$, respectively, such that $\mathbf{arr}\, F_s[x_\tau](\tau) = \mathrm{EA}_H(s,x_\tau,\tau)$ and $\mathbf{arr}\, F_t[x_\tau](\mathrm{EA}_H(s,x_\tau,\tau)) = \mathrm{EA}_H(x_\tau,t,\mathrm{EA}_G(s,x_\tau,\tau)) = \mathrm{EA}_H(s,t,\tau)$ holds (due to Lemma 3.8, 4.11, and 4.33). Together, we have $\mathbf{arr}(F_t[x_\tau] * F_s[x_\tau])(\tau) = \mathrm{EA}_H(s,t,\tau)$ with $x_\tau \in X =: \{x_1, \ldots, x_k\}$. So, $\mathbf{arr}\, f_i(\tau) = \mathbf{arr}(F_t[x_\tau] * F_s[x_\tau])(\tau) = \mathrm{EA}_H(s,t,\tau)$ for $x_\tau = x_i$ yields

$$\min(f_1, \ldots \min(f_i, \ldots \min(f_{k-1}, f_k) \ldots) \ldots) = \mathrm{TT}_H(s,t,\cdot) = \mathrm{TT}_G(s,t,\cdot)\ ,$$

which is the desired result. □

Accelerating TTP Queries with Cones. According to our experiments Algorithm 5.8 is feasible, but still not fast enough (see Section 5.6). But the TCH-based TTP query can be further accelerated by a preceding bidirectional TTP interval search (see Algorithm 5.10). The idea is to extract the relatively small subgraphs $S_\uparrow \subseteq H_\uparrow$ and $S_\downarrow \subseteq H_\downarrow$ from the predecessor graphs of the bidirectional TTP interval search. As $S_\uparrow \cup S_\downarrow$ contains enough nodes and edges to compute $\mathrm{TT}_G(s,t,\cdot)$ correctly but not too many others, Algorithm 5.8 can then be applied to $S_\uparrow \cup S_\downarrow$ for computing $\mathrm{TT}_G(s,t,\cdot)$ within much less running time.

The subgraphs $S_\uparrow$ and $S_\downarrow$ can be obtained quite easily: In case of $S_\uparrow$, we just perform a BFS on the transpose predecessor graph $(H_\uparrow(p_s))^\top$ of the forward search starting from the candidate nodes stored in X. There, it is not necessary to build $(H_\uparrow(p_s))^\top$ explicitly, because the predecessor information p_s can be used

Algorithm 5.10. An accelerated version of the TTP query on TCHs. It uses bidirectional TTP interval search to obtain an opening cone $S_\uparrow \subseteq H_\uparrow$ and a closing cone $S_\downarrow \subseteq H_\downarrow$. The more expensive TTP query described in Algorithm 5.8 is invoked afterwards, but only to run on the heavily reduced TCH substructure $S_\uparrow \cup S_\downarrow \subseteq H$, which contains only few nodes and edges that are not part of an (s,t,τ)-EA up-down-path for some $\tau \in \mathbb{R}$. Invokes *tchTtpQuery* (see Algorithm 5.8), *ttpIntervalRelax* (see Algorithm 5.6), and *extractCone* (see Algorithm 5.11) as subprocedures.

```
function $tchConeTtpQuery(s,t : V) : TTF$
    $[q_s[u], r_s[u]] := [q_t[u], r_t[u]] := [\infty,\infty]$, $p_s[u] := p_t[u] := \emptyset$ for all $u \in V$
    $[q_s[s], r_s[s]] := [q_t[t], r_t[t]] := [0,0]$
    $B := \infty$
    $X := \emptyset : Set$
    $Q_s := \{(s,0)\}, Q_t := \{(t,0)\} : PriorityQueue$
    $\Delta := t$
    while $(Q_s \neq \emptyset$ or $Q_t \neq \emptyset)$ and $\min\{Q_s.min(), Q_t.min()\} \leq B$ do
        if $Q_{\neg\Delta} \neq \emptyset$ then $\Delta := \neg\Delta$                // with $\neg s := t$ and $\neg t := s$
        $u := Q_\Delta.deleteMin()$
        if $B < \infty$ and $q_s[u] + q_t[u] \leq B$ then $X := X \cup \{u\}$
        $B := \min\{B, r_s[u] + r_t[u]\}$
        for $u \to_f v \in E_\Delta$ do                // with $E_s := E_\uparrow$ and $E_t := E_\downarrow^\top$
            $ttpIntervalRelax(u \to_f v, q_\Delta, r_\Delta, p_\Delta, Q_\Delta)$
    $(S_\uparrow, S_\downarrow) := \big(extractCone(s,X,p_s), \big(extractCone(t,X,p_t)\big)^\top\big)$
                // computes $(S_\uparrow, S_\downarrow) := \Big(\mathscr{P}_{H_\uparrow(p_s)}(s,X), \big(\mathscr{P}_{H_\downarrow^\top(p_t)}(t,X)\big)^\top\Big)$
    return $tchTtpQuery(s,t)$ using $S_\uparrow \cup S_\downarrow$ as underlying TCH structure
```

Algorithm 5.11. Computes the cone with start node s and end nodes X in the predecessor graph represented by p. The computation is actually a BFS starting from the nodes in X adding the transpose version of all touched edges to the resulting cone. The computation is very similar to Algorithm 5.2 with the difference that the BFS starts at a set $X \subseteq V$ instead of a single node.

```
function extractCone($s : Node, X : Set, p : PredInfo$) : Graph
    $(V_S, E_S) := (\emptyset, \emptyset)$                    // the result cone, initially empty
    $Q := \emptyset$ : FifoQueue
    for $x \in X$ do
        $Q.push(x)$
        mark $x$ as visited
    while $Q \neq \emptyset$ do
        $v := Q.pop()$
        foreach $u \in p[v]$ do
            add $u \to v$ to $E_S$
            if $u$ already marked as visited then continue
            $Q.push(u)$
            mark $u$ as visited
    add all nodes that are marked as visited to $V_S$
    return $(V_S, E_S)$
```

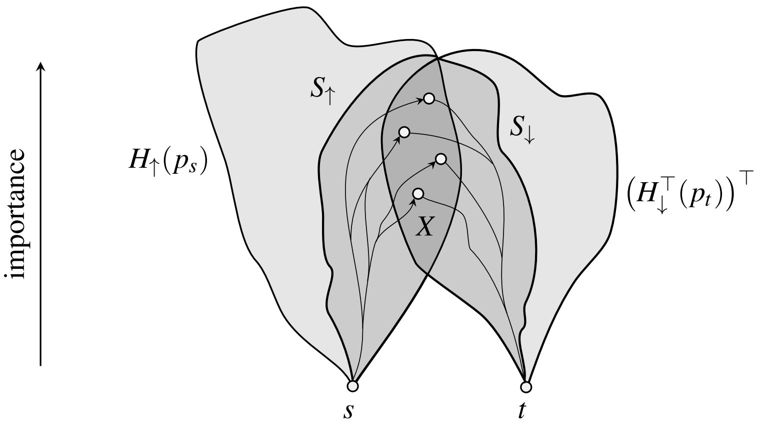


Figure 5.6. The different subgraphs of $H = H_{\uparrow} \cup H_{\downarrow}$ that occur during the accelerated version of TCH-based TTP query (see Algorithm 5.10). We look at the TCH structure H "from the side" with the more important nodes higher up. The predecessor graphs $H_{\uparrow}(p_s) \subseteq H_{\uparrow}$ and $(H_{\downarrow}^{\top}(p_t))^{\top} \subseteq H_{\downarrow}$ of forward and backward TTP interval search, respectively, have the candidate node set X in common. Together, the cones $S_{\uparrow} \subseteq H_{\uparrow}(p_s)$ and $S_{\downarrow} \subseteq (H_{\downarrow}^{\top}(p_t))^{\top}$ contain some up-down-paths from s to t. Amongst them, there is at least one (s, t, τ)-EA up-down-path for every $\tau \in \mathbb{R}$.

directly. During the BFS, a reverted version of every touched edge of $(H_\uparrow(p_s))^\top$ is added to $S_\uparrow$. Algorithm 5.11 shows the respective pseudocode. For $S_\downarrow$ all this works analogously but with a BFS on $(H_\downarrow^\top(p_t))^\top$. Note that it is not necessary to perform the transposition of

$$\mathit{extractCone}(t,X,p_t) = \mathscr{P}_{H_\downarrow^\top(p_t)}(t,X) \subseteq H_\downarrow^\top$$

in a separate step. Instead, we can modify Line 10 of Algorithm 5.11, such that $v \to u$ is added to E_S instead of $u \to v$. The subgraph $S_\uparrow \subseteq H_\uparrow$ and $S_\downarrow \subseteq H_\downarrow$ is an opening and a closing cone respectively. Figure 5.6 depicts the different subgraphs of H that occur when the TCH-based TTP query described in Algorithm 5.10 is performed.

Theorem 5.9. *Let H be a TCH generated from G and $s,t \in V$. Then, Algorithm 5.10 returns* $\mathrm{TT}_G(s,t,\cdot)$.

Proof. For every $\tau \in \mathbb{R}$ there is an up-down-path with top node x_τ in H which is a prefix-optimal (s,t,τ)-EA-path in H, as Theorem 5.5 tells us. Surely, x_τ is reached both by forward and backward search of the bidirectional TTP interval search and thus contained in X. Correspondingly, there is a (s,x_τ,τ)-EA-path $P_\tau \subseteq H_\uparrow(p_s) \subseteq H_\uparrow$ and an $(x_\tau,t,\mathrm{EA}_G(s,x_\tau,\tau))$-EA-path $P'_\tau \subseteq (H_\downarrow^\top(p_t))^\top \subseteq H_\downarrow$ (due to Lemma 4.14 and 4.36). Of course, $S_\uparrow$ and $S_\downarrow$ are an opening and closing cone, respectively, with $P_\tau \subseteq S_\uparrow$ and $P'_\tau \subseteq S_\downarrow$. As x_τ lies on an (s,t,τ)-EA up-down-path, P_τ and P'_τ together form an (s,t,τ)-EA up-down-path $R_\tau := P_\tau P'_\tau$. So, because of $R_\tau \subseteq S_\uparrow \cup S_\downarrow$, we apply Theorem 5.8 with $H_\uparrow := S_\uparrow$ and $H_\downarrow := S_\downarrow$ and are finished. □

5.3.3 Stall-on-Demand

The running time of EA and TTP queries using TCH structures can be further improved using a technique called *stall-on-demand*. It has been originally developed in the context of highway node routing [74] (for a short summary see Section 1.3.1 on page 30) and is also used with constant travel cost CHs [44]. The idea is to stop the forward or backward search at nodes where we can easily prove that H contains better paths than $H_\uparrow$ or $H_\downarrow$ alone, respectively. If this happens, we say that a node u is *stalled* regarding the forward or backward search, respectively. Here we explain, how the ideas of stall-on-demand can be applied if the travel costs are time-dependent travel times.

Stall-on-Demand for EA Queries. Stall-on-demand happens during the first phase of the TCH-based EA query (see Algorithm 5.4); that is, during the bidirectional upward search. As forward and backward search are different kinds of Dijkstra-like algorithms, stall-on-demand works a little different each.

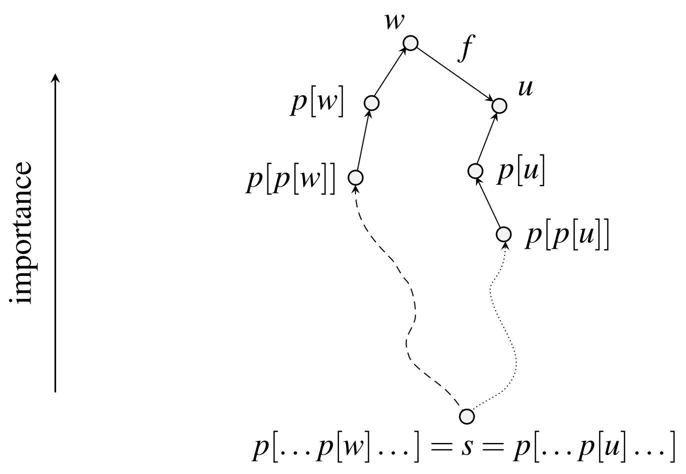


Figure 5.7. Stalling of a node u with respect to forward search during the bidirectional phase of the TCH-based EA query. Having removed u from the PQ, we check the downward edges $w \to_f u \in E_\downarrow$ and stall u if $\mathbf{arr}\, f(\tau[w]) < \tau[u]$ is fulfilled for one of these edges. This is correct, because $\mathbf{arr}\, f(\tau[w]) < \tau[u]$ implies that the path $\langle s \to \cdots \to p[p[w]] \to p[w] \to w \to_f u \rangle$ (partly drawn dashed) provides an earlies arrival at u than the path $\langle s \to \cdots \to p[p[u]] \to p[u] \to u \rangle$ (partly drawn dotted) does. We look at the TCH structure H "from the side" with the more important nodes higher up.

Stall-on-Demand Forward. With respect to the forward search, which is a time-dependent Dijkstra in $H_\uparrow$, stall-on-demand works as follows: Whenever the forward search removes a node u from the PQ (see Line 21 of Algorithm 5.4), we examine all the incoming edges of the node u in $H_\downarrow$ trying to find an arrival time which is better than $\tau[u]$. More precisely, we check whether $\mathbf{arr}\, f(\tau[w]) < \tau[u]$ holds for some edge $w \to_f u \in E_\downarrow$. If this is the case, the node u is *stalled* with respect to the forward search. This means the edges of u in $H_\uparrow$ are not relaxed. It is important to note that the result of the EA query is not affected if nodes are stalled in this manner.

Lemma 5.10. *Stalling a node with respect to forward search preserves the correctness of TCH-based EA queries.*

Proof. The existence of an edge $w \to_f u \in E_\downarrow$ with $\mathbf{arr}\, f(\tau[w]) < \tau[u]$ proves that the path

$$P_{su} := \big\langle s = p[\dots p[u]\dots] \to \cdots \to p[u] \to u \big\rangle \subseteq H_\uparrow$$

from s to u found by the forward search can not be an (s,u,τ_0)-EA-path in H (although it is an (s,u,τ_0)-EA-path with respect to $H_\uparrow$). Figure 5.7 illustrates this. As a consequence, P_{su} can not be prefix of an (s,t,τ_0)-EA up-down-path in H with prefix-optimal upward part. But this is exactly the kind of EA up-down-paths found by Algorithm 5.4. So, Algorithm 5.4 stays correct if stall-on-demand is applied, because stall-on-demand only prevents upward paths that are not prefix-optimal. □

If a node u is successfully stalled, we can propagate the stalling to further nodes as a better path to u may be part of better paths to some other nodes. The propagation works in the manner of a BFS in $H_\uparrow$ and stops at nodes we fail to stall, that are already stalled, or that are not reached by the forward search yet. This way, we are able to examine paths

$$P_{swuv} := \left\langle s = p[\dots p[w] \dots] \to \cdots \to p[w] \to w \to_f u \to \cdots \to v \right\rangle$$

with $\langle s = p[\dots p[w] \dots] \to \cdots \to p[w] \to w\rangle, \langle u \to \cdots \to v\rangle \subseteq H_\uparrow$, and $w \to_f u \in E_\downarrow$, which are no up-down-paths and can not be examined by bidirectional upward search, hence. Whenever a node v is removed from the PQ while $\mathbf{arr}\, f_{P_{swuv}}(\tau_0) < \tau[v]$ is fulfilled, then we know that the path

$$\left\langle s = p[\dots p[v] \dots] \to \cdots \to p[v] \to v \right\rangle \subseteq H_\uparrow \, ,$$

which is the final path from s to v found by the forward search, can not be part of an (s,t,τ_0)-EA up-down-path in H with a prefix-optimal upward part.

To store information about paths P_{swuv}, which are detected by stalling and propagation, we maintain a value $\tau_{\text{stall}}[v]$ for every node v and set $\tau_{\text{stall}}[v] := \mathbf{arr}\, f_{P_{swuv}}(\tau_0)$ whenever a node v is stalled directly or by propagation. Whenever, a node u is removed from the PQ, we check whether $\tau_{\text{stall}}[u] < \tau[u]$ is fulfilled. If this is the case, then u is already stalled because of propagation. Otherwise, we try to stall u by examining the incoming edges of u in $H_\downarrow$.

Algorithm 5.12 shows the TCH-based EA query, applying stall-on-demand with respect to both forward and backward search. The pseudocode of stalling and propagation is factored out and can be found in Algorithm 5.13 and 5.14. Algorithm 5.13 regards the forward search and Algorithm 5.14 the backward search. The EA query specified in Algorithm 5.12 is the same as in Algorithm 5.4. The only difference is that the Lines 18, 23, and 24 are inserted. Figure 5.8 illustrates how stalling and propagation work with respect to the forward search.

Corollary 5.11. *Propagating the stalling of a node with respect to forward search preserves the correctness of TCH-based EA queries.*

Stall-on-Demand Backward. During the backward search, which is a backward TTP interval search in $H_\downarrow$ (see Section 4.3.2), stall-on-demand works similar as in case of the forward search but with the difference that we do not have exact travel times. Instead, we prune in terms of upper and lower bounds. Other differences to the forward search are that the roles of $H_\uparrow$ and $H_\downarrow$ must be exchanged and that

Algorithm 5.12. TCH-based EA query as in Algorithm 5.4 but augmented with stall-on-demand. Additionally to *tdRelax* (see Algorithm 5.5) and *ttpIntervalRelax* (see Algorithm 5.6), it invokes *stalledForward* (see Algorithm 5.13) and *stalledBackward* (see Algorithm 5.14) as subprocedure, which perform stalling and propagation for forward and backward search, respectively.

```
1  function $tchEaQueryWithSoD(s,t : Node, \tau_0 : \mathbb{R}) : Path$
2      $\tau[u] := \infty$ for all $u \in V$, $\tau[s] := \tau_0$,
3      $[q[u], r[u]] := [\infty, \infty]$ for all $u \in V$, $[q[t], r[t]] := [0,0]$
4      $p_s[u] := \perp$, $p_t[u] := \emptyset$ for all $u \in V$
5      $B := \infty$                                   // upper bound of EA time
6      $X := \emptyset : Set$                          // set of candidate nodes
7      function $downwardSearch() : Path$
8          $\tau_{\mathrm{down}}[u] := \infty$ for all $u \in V$
9          $Q_{\mathrm{down}} := \emptyset : PriorityQueue$      // PQ for downward search
10         foreach $u \in X$ do
11             if $\tau_{\mathrm{down}}[u] < \infty$ and $\tau[u] + q[u] \leq B$ then
12                 $\tau_{\mathrm{down}}[u] := \tau[u]$, $Q_{\mathrm{down}}.insert(u, \tau[u])$
13         while $Q_{down} \neq \emptyset$ do
14             $u := Q_{\mathrm{down}}.deleteMin()$
15             if $u = t$ then return $\langle s = p_s[\dots p_s[t] \dots] \to \cdots \to p_s[t] \to t \rangle$
16             for $v \in p_t[u]$ do $tdRelax(u \to_f v, \tau_{\mathrm{down}}, p_s, Q_{\mathrm{down}})$
17     $Q_s := \{(s, \tau_0)\}$, $Q_t := \{(t, 0)\} : PriorityQueue$
18     $\tau_{\mathrm{stall}}[u] := r_{\mathrm{stall}}[u] := \infty$ for all $u \in V$    // propagation information for stalling
19     $\Delta := t$
20     while $(Q_s \neq \emptyset$ or $Q_t \neq \emptyset)$ and $\min\{Q_s.min(), Q_t.min()\} \leq B$ do
21         if $Q_{\neg\Delta} \neq \emptyset$ then $\Delta := \neg\Delta$     // change of direction: $\neg s := t$ and $\neg t := s$
22         $u := Q_\Delta.deleteMin()$
23         if $\Delta = s$ and $stalledForward(u, \tau, \tau_{\mathrm{stall}})$ then continue
24         else if $\Delta = t$ and $stalledBackward(u, q, r_{\mathrm{stall}})$ then continue
25         if $B < \infty$ and $\tau[u] + q[u] \leq B$ then $X := X \cup \{u\}$
26         $B := \min\{B, \tau[u] + r[u]\}$
27         for $u \to_f v \in E_\Delta$ do                 // with $E_s := E_\uparrow$ and $E_t := E_\downarrow^\top$
28             if $\Delta = s$ then $tdRelax(u \to_f v, \tau, p_s, Q_s)$
29             else $ttpIntervalRelax(u \to_f v, q, r, p_t, Q_t)$
30     if $B = \infty$ then return $\langle\rangle$
31     return $downwardSearch()$
```

Algorithm 5.13. Tries to stall a given node x with respect to forward search during the bidirectional phase of TCH-based EA query. In case of success, the stalling of x is propagated to further nodes in the manner of a BFS on $H_\uparrow$. The propagation is pruned at every node that is not reached by the forward search yet, already stalled, or if the stalling of this node fails.

```
function stalledForward($x : V,\ \tau, \tau_{\text{stall}} : Reference$) : Bool
    procedure propagateStallingForward($x : V,\ \tau, \tau_{\text{stall}} : Reference$)
        $Q := \{x\}$ : FifoQueue                      // propagate in manner of a BFS
        while $Q \neq \emptyset$ do
            $u := Q.popFront()$
            for $u \to_f v \in E_\uparrow$ do          // prune if unreached, already stalled,...
                $\tau_{\text{new}} := \mathbf{arr}\, f(\tau_{\text{stall}}[u])$          // ...or stalling fails
                if $\tau[v] = \infty$ or $\tau_{\text{stall}}[v] < \infty$ or $\tau[v] \le \tau_{\text{new}}$ then continue
                $\tau_{\text{stall}}[v] := \tau_{\text{new}}$
                $Q.pushBack(v)$
    if $\tau_{\text{stall}}[x] < \tau[x]$ then return true               // check if already stalled
    foreach $u \to_f x \in E_\downarrow$ do
        if $\mathbf{arr}\, f(\tau[u]) < \tau[x]$ then
            $\tau_{\text{stall}}[x] := \mathbf{arr}\, f(\tau[u])$
            propagateStallingForward($x, \tau, \tau_{\text{stall}}$)
            return true
    return false
```

a node label is not necessarily final when a node is removed from the PQ. The respective factored out pseudocode can be found in Algorithm 5.14.

Whenever the backward search removes a node u from the PQ, we examine all outgoing edges of u in $H_\uparrow$ to find out whether H contains a better path from u to t than $(H_\downarrow^\top(p_t))^\top \subseteq H_\downarrow$. To do so, we check whether $q[u] > \max f + r[w]$ holds for some $u \to_f w \in E_\uparrow$. If this is the case, we know that no (u,t,τ)-EA-path with respect to H is contained in the current $(H_\downarrow^\top(p_t))^\top$ for any $\tau \in \mathbb{R}$, because a path $\langle w \to \cdots \to t\rangle \subseteq (H_\downarrow^\top(p_t))^\top \subseteq H_\downarrow$ exists with

$$\begin{aligned}\mathrm{TT}_{(H_\downarrow^\top(p_t))^\top}(u,t,\tau) \ge q[u] > \max f + r[w] &\\ &\ge f(\tau) + f_{\langle w\to\cdots\to t\rangle}(f(\tau)+\tau)\\ &= f_{\langle w\to\cdots\to t\rangle} * f(\tau)\\ &= f_{\langle u\to_f w\to\cdots\to t\rangle}(\tau)\\ &\ge \mathrm{TT}_H(u,t,\tau)\end{aligned}$$

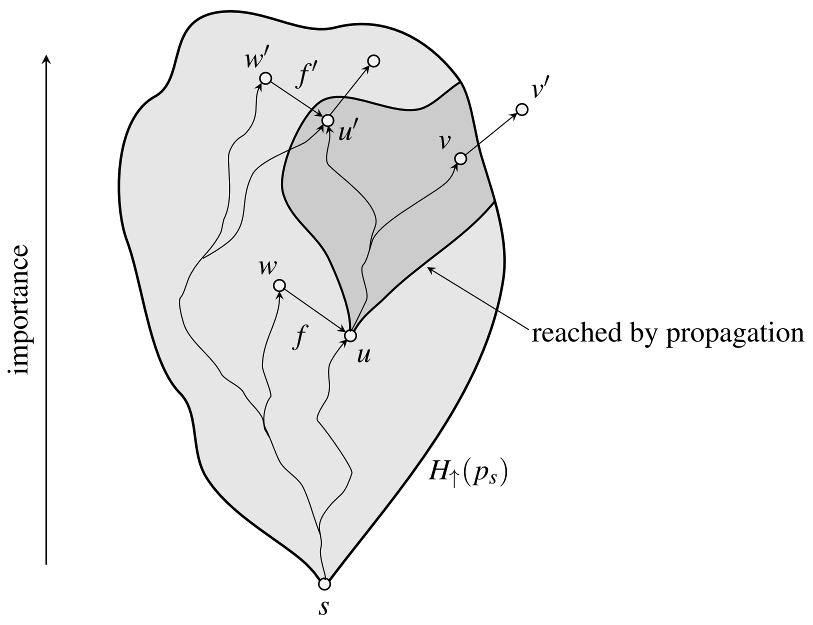


Figure 5.8. Illustration of stalling and propagation with respect to the forward search during the first phase of the TCH-based EA query (i.e., during the bidirectional search). The predecessor graph $H_{\uparrow}(p_s)$ of the forward search is depicted in light gray. The node u is stalled because a downward edge $w \to_f u \in E_{\downarrow}$ is present with $\mathbf{arr}\, f(\tau[w]) < \tau[u]$. The stalling of u is propagated upward in H. The subgraph of $H_{\uparrow}$ reached by the propagation (which is actually a BFS in $H_{\uparrow}$ starting from u) is depicted in dark gray. The propagation reaches the node v but not the node v', because v' is not reached by the forward search yet. Note that pruning the propagation at unreached nodes makes it a BFS in $H_{\uparrow}(p_s)$. The propagation is also pruned at node u', although u' has a successor in in $H_{\uparrow}(p_s)$. This is because u' is already stalled; namely, via the edge $w' \to_{f'} u' \in E_{\downarrow}$ with $\mathbf{arr}\, f'(\tau[w']) < \tau[u']$. Note that the propagation is also pruned at nodes that fail to be stalled by the propagation.

for all $\tau \in \mathbb{R}$. This follows from $r[w] \geq \overline{\mathrm{TT}}_{H_{\downarrow}}(w,t) \geq f_{\langle w \to \cdots \to t \rangle}(\sigma)$ for all $\sigma \in \mathbb{R}$ as well as the third statement of Lemma 4.55, where backward TTP interval search is considered as a special case of backward approximate CP search. This means that the current $H_{\uparrow} \cup (H_{\downarrow}^{\top}(p_t))^{\top}$ does not contain an EA up-down-path from s to t in H running through u. So, the node u can be stalled with respect to the backward search, unless $q[u]$ gets less or equal $\max f + r[w]$. So, the edges $u \to v \in E_{\downarrow}^{\top}$ are not relaxed at this removal of u from the PQ. If $q[u]$ really gets less or equal $\max f + r[w]$, then u must have been reinserted into the PQ with a new chance to relax its edges.

Corollary 5.12. *Stalling a node with respect to backward search preserves the*

Algorithm 5.14. Tries to stall a given node x with respect to backward search during the bidirectional phase of TCH-based EA query (see Algorithm 5.4 and 5.12). In case of success, the stalling of x is propagated to further nodes in the manner of a BFS on $H_\downarrow^\top$. Can also be used to realize stall-on-demand with respect to the backward TTP interval search performed during the accelerated TTP query (see Algorithm 5.10).

```
function $stalledBackward(x : V,\ q, r_{\text{stall}} : Reference) : Bool$
    procedure $propagateStallingBackward(x : V,\ q, r_{\text{stall}} : Reference\ )$
        $Q := \{x\} : FifoQueue$                                  // propagate in manner of a BFS
        while $Q \neq \emptyset$ do
            $u := Q.popFront()$
            for $u \to_f v \in E_\downarrow^\top$ do                   // prune if unreached, already stalled,...
                $r_{\text{new}} := \max f + r_{\text{stall}}[u]$                        // ...or stalling fails
                if $q[v] = \infty$ or $r_{\text{stall}}[v] < \infty$ or $q[v] \leq r_{\text{new}}$ then continue
                $r_{\text{stall}}[v] := r_{\text{new}}$
                $Q.pushBack(v)$
    if $r_{\text{stall}}[x] < q[x]$ then return $true$                      // check if already stalled
    foreach $x \to_f u \in E_\uparrow$ do
        if $\max f + r[u] < q[x]$ then
            $r_{\text{stall}}[x] := \max f + r[u]$
            $propagateStallingBackward(x, q, r_{\text{stall}})$
            return $true$
    return $false$
```

correctness of TCH-based EA queries.

Stalling of nodes with respect to backward search can also be propagated to further nodes, just like in case of forward search. To do so, we perform a BFS in $H_\downarrow^\top$ starting from a node u when the node u gets stalled (see Line 2 to 10 in Algorithm 5.14). Again the BFS stops at nodes we fail to stall, that are already stalled, or that are not reached by the backward search yet. Analogously to stall-on-demand with respect to forward search, the propagation examines paths

$$P_{vuwt} := \left\langle v \to_{g_1} \cdots \to_{g_k} u \to_f w \to \cdots \to t \right\rangle$$

with $\langle v \to \cdots \to u\rangle, \langle w \to \cdots \to t\rangle \subseteq H_\downarrow$ and $u \to_f w \in E_\uparrow$. Such paths are no up-down-paths and can not be examined by bidirectional upward search hence. To store informations about paths like P_{vuwt}, we maintain a value $r_{\text{stall}}[v]$ for every node $v \in V$. When the backward search removes the node v with $q[v] > r_{\text{stall}}[v]$

from the PQ, we know

$$\begin{aligned}\mathrm{TT}_{(H_\downarrow^\top(p_t))^\top}(v,t,\tau) \geq q[v] > r_{\mathrm{stall}}[v] \\ &\geq \max g_1 + \cdots + \max g_k + \max f + r[w] \\ &\geq f_{\langle w \to \cdots \to t\rangle} * f * g_k * \cdots * g_1(\tau) \\ &= f_{P_{vuwt}}(\tau) \\ &\geq \mathrm{TT}_H(v,t,\tau)\end{aligned}$$

holds for all $\tau \in \mathbb{R}$, thanks to Equation (3.8) as well as the third statement of Lemma 4.55 with backward TTP interval search considered as special case of backward approximate CP search. So, $(H_\downarrow^\top(p_t))^\top$ does not contain a (v,t,τ)-EA-path with respect to H for any $\tau \in \mathbb{R}$. This means, v can not lie on an the downward part of an EA up-down-path from s to t unless $q[v]$ gets less or equal $r_{\mathrm{stall}}[v]$ again. This means, the backward search can be pruned at v; at least regarding the current removal of v from the PQ.

Corollary 5.13. *Propagating the stalling of a node with respect to backward search preserves the correctness of EA querying with TCH structures.*

Note that, in case of the backward search, stall-on-demand may fail to stall a node v, even if $\mathrm{TT}_{H_\downarrow}(v,t,\tau) > \mathrm{TT}_H(v,t,\tau)$ holds for all $\tau \in \mathbb{R}$. This can happen, because we only work in terms of lower and upper bounds here. If, for example, $\min \mathrm{TT}_{H_\downarrow}(v,t,\cdot) \leq \max \mathrm{TT}_H(v,t,\cdot)$ holds, then v is not stalled (though it would not destroy the correctness of the EA query to do so).

Stall-on-Demand for TTP Queries. Stall-on-demand can not only be used with EA queries, but also to TTP queries. More precisely, we apply stall-on-demand to the accelerated version of TTP query that uses bidirectional TTP interval search to compute cones (see Algorithm 5.10). There, we apply stall-on-demand only to the bidirectional TTP interval search and not to the subsequent bidirectional TTP search. Regarding the backward search, this is the same as in case of EA queries, where the backward search is a backward TTP interval search, too.

To obtain a procedure for stalling and propagation that works with the forward search (which now is a TTP interval search instead of a time-dependent Dijkstra), we adapt the technique used with the backward search. This way, we are sometimes able to examine paths

$$\langle s \to \cdots \to w \to_f u \to_{g_1} \cdots \to_{g_k} v\rangle$$

with $\langle s \to \cdots \to w\rangle, \langle u \to \cdots \to v\rangle \subseteq H_\uparrow$ and $w \to_f v \in E_\downarrow$, which can not be detected by a bidirectional upward search in H. This enables us to prune the search at v, if

$$q_s[v] > q[w] + \max f + \max g_1 + \cdots + \max g_k$$

is fulfilled when the forward search removes v from the PQ. To check this condition, we maintain a value $r_{s,\mathrm{stall}}[v]$ for every node v and look whether $q_s[v] > r_{s,\mathrm{stall}}[v]$ holds true when v is removed from the PQ. Analogous to the backward search, we argue that this implies $\mathrm{TT}_{H_\uparrow(p)}(s,v,\tau) \geq q_s[v] > r_{s,\mathrm{stall}}[v] \geq \mathrm{TT}_H(s,v,\tau)$ for all $\tau \in \mathbb{R}$ (due to Lemma 4.19 with TTP interval search considered as special case of approximate TTP search). This means that no path in the current $H_\uparrow(p_s)$ is an EA-path from s to v with respect to H. So, the correctness of the TTP query algorithm is preserved.

That all the variants of stalling and propagation described in this section are well-behaved, can be summarized as follows.

Theorem 5.14. *Stall-on-demand as described in this section does not affect the correctness of exact TCH-based EA and TTP queries.*

Different Flavors of Propagation. As explained above, we prune the propagation of the stalling at nodes that are not reached yet, that are already stalled, or that we fail to stall. But, in principle, we could continue the propagation even if stalling fails at node. However, propagating the stalling to a large number reachable nodes may take so much time that the acceleration achieved by the stalling is eaten up. This is an obvious tradeoff. Note that we have not examined systematically how far the stalling should be propagated. Two implementations of constant travel cost CHs provided by Luxen and Vetter [58], for example, even propagate the stalling only one hop.

5.4 Exact Space Efficient Querying

With the techniques described in Section 5.3, TCHs allow very fast answering of EA and TTP queries. Unfortunately, TCH structures need a lot of memory compared to the data structure needed to run time-dependent Dijkstra. We overcome this problem by the careful use of approximation. More precisely, we generate approximate versions of the TCH structures which need much less space (see Section 5.4.1). For EA queries we get a moderate slowdown this way, but the results of the computations are still exact (see Section 5.4.2). For profile queries we also get exact results but we additionally obtain a further speedup (see Section 5.4.3).

5.4.1 Approximate Time-Dependent Contraction Hierarchies

To save memory, we use *approximate TCHs (ATCHs)* and *Min-Max-TCHs*. Although these variants of TCHs contain partly approximated data, they can be used to compute exact results. An *ATCH structure* with *relative error* $\varepsilon > 0$ is generated from a given TCH structure H as follows: For all original edges $u \rightarrow_f v$ (i.e., $u \rightarrow_f v \in E \subseteq E_H$), nothing happens. All other edges $u \rightarrow_f v$ (i.e., $u \rightarrow_f v \in E_H \setminus E$) are shortcuts and their TTFs f are replaced by TTFs $\overline{f}$ with

$$\forall \tau \in \mathbb{R} : f(\tau) \leq \overline{f}(\tau) \leq (1+\varepsilon)f(\tau) \, . \tag{5.18}$$

The TTF $\overline{f}$ is an upper bound of f. Implicitly, $\overline{f}$ also represents a lower bound

$$\underline{f} : \tau \mapsto \overline{f}(\tau)/(1+\varepsilon) \, . \tag{5.19}$$

For all original edges $u \rightarrow_f v \in E$ we set $\overline{f} := \underline{f} := f$. To denote an edge $u \rightarrow v$ with lover bound $\underline{f}$ and upper bound $\overline{f}$ of an ATCH structure, we sometimes write $u \rightarrow_{(\underline{f},\overline{f})} v$. Note that we often write $u \rightarrow_f v$ for edges of ATCH structures, although the exact TTF f is no longer materialized.

Usually, $|\overline{f}|$ is considerably smaller than $|f|$. Correspondingly, an ATCH structure needs considerably less memory than the corresponding TCH structure (see Section 5.6). To compute $\overline{f}$ from an exact TTF f, we use an implementation of an efficient geometric algorithm by Imai and Iri [52], which has been provided by Neubauer [66] (it is already mentioned in Section 4.2.4). The Imai-Iri algorithm yields an $\overline{f}$ of minimal complexity $|\overline{f}|$ for a given ε in time $O(|f|)$. The computed $\overline{f}$ may violate the FIFO property, but this can be repaired in $O(|\overline{f}|)$ time. If $\overline{f}$ fulfills the FIFO property, then $\underline{f} = (1+\varepsilon)^{-1}\overline{f}$ also does.

A *Min-Max-TCH structure* is an extreme case of an ATCH structure. For edges $u \rightarrow_f v \in E_H \setminus E$ we set $\underline{f} :\equiv \min f$ and $\overline{f} :\equiv \max f$, which means we only store a pair of numbers in this case. Min-Max-TCHs need even less memory than ATCHs. The query times, in contrast, get larger. Especially TTP queries are significantly slower with Min-Max-TCHs, as our experiments show (see Section 5.6).

Note that we do not apply approximation during preprocessing. In this work approximation is only applied *after* preprocessing. Thus, ATCHs and Min-Max-TCHs are generated from complete exact TCHs only. However, applying approximation during preprocessing is an interesting idea. First, TCH preprocessing could be further accelerated this way, because approximate TTP search runs much faster than exact TTP search. If an approximate TTP search already shows that no shortcuts is needed, then the slow exact TTP search can be omitted. Second, TCH preprocessing could be made more robust. The idea is to store shortcuts with upper and lower bounds instead of exact TTFs even during preprocessing. This can prevent that necessary shortcuts are mistakenly not added because of rounding errors, as decisions in terms of lower and upper bounds are more conservative. Especially for very large road networks, rounding errors may become a problem. It must be noted, however, that more unnecessary shortcuts may be inserted this way. If the number of shortcuts increases too much, than preprocessing and querying may get significantly slower. Whether this happens or not, must be found out experimentally of course.

5.4.2 Earliest Arrival Queries

Given an ATCH structure H with relative error ε we want to compute the *exact* value of $\mathrm{EA}_G(s,t,\tau_0)$ and a corresponding EA path. To do so, we use Algorithm 5.15, which works in three phases. Actually, to obtain an exact result, we had to perform a time-dependent Dijkstra in G, because we only have exact TTFs in $G \subseteq H$. But this would be slow, so the idea is to restrict the time-dependent Dijkstra to a very small corridor $S \subseteq G$. For this reason we first use the ATCH structure H to perform several approximate Dijkstra-like searches one after another to select and successively build and thin out a subgraph of H (phases 1 and 2). Then, we expand all shortcuts in that subgraph to obtain the corridor S and perform the time-dependent Dijkstra only there (phase 3). Note that there are two versions of the third phase. They are specified in Algorithm 5.18 and 5.19 each. As usual, the different edge relaxations have been factored out to prevent too long pseudocode (see Algorithms 5.6, 5.16, and 5.17).

Phase 1: Bidirectional Search. At first, we perform a bidirectional upward search where the forward search is an EA interval search (see Algorithm 4.5) starting from s, and the backward search is a backward TTP interval search (see Section 4.3.2) starting from t. All candidate nodes (i.e., the nodes where both searches meet) are stored in the candidate set X. Due to Lemma 4.31 and 4.36, the union of cones

$$S^X := \mathscr{P}_{H_\uparrow(p_s)}(s,X) \cup \left(\mathscr{P}_{H_\downarrow^\top(p_t)}(t,X)\right)^\top \subseteq H$$

contains an (s,t,τ_0)-EA up-down-path. So, we could skip phase 2 and continue directly with phase 3; that is with expanding all shortcuts in S^X and then running a time-dependent Dijkstra on the resulting corridor $S \subseteq G$. However, the backward search is a relatively rough approximation. So, we expect that $H_\downarrow^\top(p_t)$ and X—and thus S_X and S—are larger than necessary, which means that phase 3 needs more time than necessary. To remedy this, we perform the phase 2 to thin out S^X.

Algorithm 5.15. EA query using an ATCH H with relative error $\varepsilon > 0$. The subprocedure *corridorDijkstra* is either instantiated with *corridorDijkstraExpandFully* (see Algorithm 5.18) or *corridorDijkstraEoD* (see Algorithm 5.19). Edge relaxations are done by the subprocedures *eaIntervalRelax* (see Algorithm 5.16), *ttpIntervalRelax* (see Algorithm 5.6), and *ldIntervalRelax* (see Algorithm 5.17).

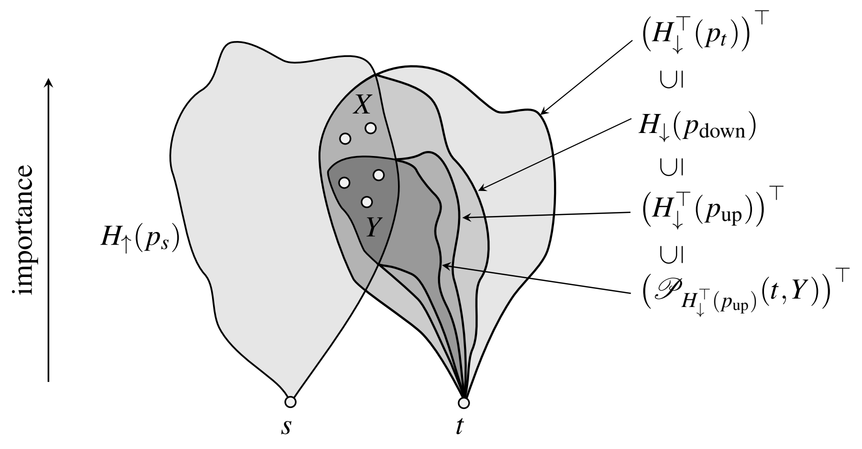
```
function $atchEaQuery(s,t : V, \tau_0 : \mathbb{R}) : Path$
    $[q_\Delta[u], r_\Delta[u]] := [\infty,\infty]$, $p_\Delta[u] := \emptyset$ for all $u \in V$, $\Delta \in \{s,t,\text{down},\text{up}\}$
    $[q_s[s], r_s[s]] := [\tau_0, \tau_0]$, $[q_t[t], r_t[t]] := [0,0]$
    $X := Y := \emptyset : Set$
    procedure $bidirectionalSearch()$
        $Q_s := \{(s,\tau_0)\}$, $Q_t := \{(t,0)\} : PriorityQueue$
        $B := \infty$, $\Delta := t$
        while $(Q_s \neq \emptyset$ or $Q_t \neq \emptyset)$ and $\min\{Q_s.min(), Q_t.min()\} \leq B$ do
            if $Q_{\neg\Delta} \neq \emptyset$ then $\Delta := \neg\Delta$                // with $\neg s := t$, $\neg t := s$
            $u := Q_\Delta.deleteMin()$
            if $B < \infty$ and $q_s[u] + q_t[u] \leq B$ then $X := X \cup \{u\}$
            $B := \min\{B, r_s[u] + r_t[u]\}$
            for $u \rightarrow_{(\underline{f},\overline{f})} v \in E_\Delta$ do                // with $E_s := E_\uparrow$, $E_t := E_\downarrow^\top$
                if $\Delta = s$ then $eaIntervalRelax(u \rightarrow_{(\underline{f},\overline{f})} v, q_s, r_s, p_s, Q_s)$
                else $ttpIntervalRelax(u \rightarrow_{(\underline{f},\overline{f})} v, q_t, r_t, p_t, Q_t)$
    procedure $downwardSearch()$
        $Q := \emptyset : PriorityQueue$
        foreach $u \in X$ with $r_s[u] + r_t[u] \leq B$ do
            $[q_\text{down}[u], r_\text{down}[u]] := [q_s[u], r_s[u]]$, $Q.insert(u, q_\text{down}[u])$
        while $Q \neq \emptyset$ do
            $u := Q.deleteMin()$
            for $v \in p_t[u]$ do $eaIntervalRelax(u \rightarrow_{(\underline{f},\overline{f})} v, q_\text{down}, r_\text{down}, p_\text{down}, Q)$
    procedure $upwardSearch()$
        $[q_\text{up}[t], r_\text{up}[t]] := [q_\text{down}[t], r_\text{down}[t]]$
        $Q := \{(t, q_\text{up}[t])\} : MaxPriorityQueue$
        while $Q \neq \emptyset$ do
            $u := Q.deleteMax()$
            if $r_\text{up}[u] < \infty$ then $Y := Y \cup \{u\}$
            for $v \in p_\text{down}[u]$ do
                $ldIntervalRelax(u \rightarrow_{(\underline{f},\overline{f})} v, q_\text{up}, r_\text{up}, p_\text{up}, Q, q_\text{down}, r_\text{down})$
    $bidirectionalSearch()$, $downwardSearch()$, $upwardSearch()$
    return $corridorDijkstra(s, t, \tau_0, p_s, Y, p_\text{up})$
```

Algorithm 5.16. Edge relaxation procedure as in EA interval search (see Algorithm 4.5). The *Reference* parameters q, r, p, Q are necessary to provide context information of the calling search algorithm.

```
procedure eaIntervalRelax($u \rightarrow_{(\underline{f},\overline{f})} v$ : Edge, $q, r, p, Q$ : Reference)
    $[q_{\text{new}}, r_{\text{new}}] := \big[\mathbf{arr}\,\underline{f}(q[u]), \mathbf{arr}\,\overline{f}(r[u])\big]$
    if $q_{\text{new}} > r[v]$ then return
    if $r_{\text{new}} < q[v]$ then $p[v] := \emptyset$
    $p[v] := \{u\} \cup p[v]$
    if $q_{\text{new}} \geq q[v]$ and $r_{\text{new}} \geq r[v]$ then return
    $[q[v], r[v]] := \big[\min\{q[v], q_{\text{new}}\},\ \min\{r[v], r_{\text{new}}\}\big]$
    if $v \notin Q$ then $Q.insert(v, q[v])$
    else $Q.updateKey(v, q[v])$
```

Algorithm 5.17. Edge relaxation procedure as in LD interval search (see Algorithm 4.6). The *Reference* parameters q, r, p, Q are necessary to provide context information of the calling search algorithm.

```
procedure ldIntervalRelax($u \rightarrow_{(\underline{f},\overline{f})} v$ : Edge, $q, r, p, Q, q_{\text{down}}, r_{\text{down}}$ : Reference)
    $[q_{\text{new}}, r_{\text{new}}] := \big[\min \mathbf{dep}\,\overline{f}(q[u]),\ \max \mathbf{dep}\,\underline{f}(r[u])\big]$
    if $r_{\text{new}} < q_{\text{down}}[v]$ or $q_{\text{new}} > r_{\text{down}}[v]$ then return
    $[q_{\text{new}}, r_{\text{new}}] := \big[\max\{q_{\text{new}}, q_{\text{down}}\},\ \min\{r_{\text{new}}, r_{\text{down}}\}\big]$
                                  // exploit arrival times computed by downward search
    if $r_{\text{new}} < q[v]$ then return
    if $q_{\text{new}} > r[v]$ then $p[v] := \emptyset$
    $p[v] := \{u\} \cup p[v]$
    if $q_{\text{new}} \leq q[v]$ and $r_{\text{new}} \leq r[v]$ then return
    $[q[v], r[v]] := \big[\max\{q[v], q_{\text{new}}\},\ \max\{r[v], r_{\text{new}}\}\big]$
    if $v \notin Q$ then $Q.insert(v, q[v])$
    else $Q.updateKey(v, q[v])$
```



$\left(H_\downarrow^\top(p_t)\right)^\top$
$\cup|$
$H_\downarrow(p_{\text{down}})$
$\cup|$
$\left(H_\downarrow^\top(p_{\text{up}})\right)^\top$
$\cup|$
$\left(\mathscr{P}_{H_\downarrow^\top(p_{\text{up}})}(t,Y)\right)^\top$
importance
$H_\uparrow(p_s)$
X
Y
s
t

Figure 5.9. How the ATCH-based EA query (see Algorithm 5.15) successively refines subgraphs in $H_\downarrow$. The first subgraph is the transpose predecessor graph $(H_\downarrow^\top(p_t))^\top \subseteq H_\downarrow$ of the backward search, which is build during the first phase (i.e., during bidirectional search). The second subgraph $H_\downarrow(p_{\text{down}})$ is the predecessor graph of the downward search, which is performed at the beginning of the second phase (i.e., the thinning) starting from the nodes in X and running in $(H_\downarrow^\top(p_t))^\top$. The third subgraph is the transpose predecessor graph $(H_\downarrow^\top(p_{\text{up}}))^\top$ of the upward search. The nodes in X that are also reached by the upward search form the smaller candidate set $Y \subseteq X$. The first step of the third phase performs a BFS in $(H_\downarrow^\top(p_{\text{up}}))^\top$ that starts from the nodes in Y to compute the subgraph $(\mathscr{P}_{H_\downarrow^\top(p_{\text{up}})}(t,Y))^\top$.

Phase 2: Thinning. Starting from the candidate nodes in X we perform a downward search, which is an EA interval search on $(H_\downarrow^\top(p_t))^\top \subseteq H_\downarrow$. This yields an arrival interval $[q_{\text{down}}[t], r_{\text{down}}[t]] \ni \text{EA}_G(s,t,\tau_0)$. Having done that, we use this arrival interval to perform an upward search: an LD interval search starting from t that runs on $(H_\downarrow(p_{\text{down}}))^\top$. The upward search reaches nodes that have already been reached by the forward search during the first phase. These meeting points are again candidate nodes that we store in the set $Y \subseteq X$. Note that the downward and the upward search have much more accurate information at hand than the backward search had during the first phase. Also, the check in Line 3 of Algorithm 5.17 should rule out several nodes. Line 4 exploits the arrival time information computed by the downward search. The predecessor graph $H_\downarrow^\top(p_{\text{up}})$ should contain less edges than $H_\downarrow^\top(p_t)$ and the set Y should be smaller than X. Thus,

$$S^Y := \mathscr{P}_{H_\uparrow(p_s)}(s,Y) \cup \left(\mathscr{P}_{H_\downarrow^\top(p_{\text{up}})}(t,Y)\right)^\top \subseteq S^X \subseteq H ,$$

which is a thinned out version of S^X, should contain significantly less edges than S^X. Figure 5.9 shows the different subgraphs of $H_\downarrow$ that occur during the first and the second phase of the ATCH-based EA query. Every of these successively obtained subgraphs is expected to be thinner than the previous one.

Phase 3: Corridor Dijkstra. Both versions of the third phase start with constructing the corridor $S_Y \subseteq H$ (see Line 2 in Algorithm 5.18 and Line 2 in Algorithm 5.19). This is actually the same process as extracting the cones $S_\uparrow$ and $S_\downarrow$ as in case of the TCH-based TTP query (see Algorithm 5.10): Starting from the nodes in the candidate set Y, we perform a BFS on $(H_\uparrow(p_s))^\top$ and on $(H_\downarrow^\top(p_{\text{up}}))^\top$, respectively, and copy all touched edges to $S^Y \subseteq H$. Figure 5.10 illustrates that $S^Y \subseteq H$ is expected to be thinner than $S^X \subseteq H$, because Y is expected to be smaller than X and $(H_\downarrow^\top(p_{\text{up}}))^\top$ is expected to be thinner than $(H_\downarrow^\top(p_t))^\top$.

Now, we can recursively expand all shortcuts in S_Y to obtain a subgraph $S \subseteq G$ that does not contain any shortcuts. This is what happens in Algorithm 5.18. Of course, all TTFs in S are exact, because all edges edges of S lie in G and only edges in $H \setminus G$ have inexact TTFs. So, a time-dependent Dijkstra search in S is enough to compute an (s,t,τ_0)-EA path in G.

Although this works fast, we can still be a little faster: The idea is to perform the time-dependent Dijkstra directly on S_Y and to expand shortcuts only *on demand*. This is what happens in Algorithm 5.19. More precisely, whenever we relax a shortcut edge

$$u \rightarrow_{(\underline{f},\overline{f})} v \in E_H \setminus E ,$$

for which we do not have exact TTFs, we only expand *this* shortcut and only for the departure time $\tau[u]$ (where $\tau[u]$ is the label of u with respect to the time-dependent Dijkstra). The resulting original path $\langle u = w_1 \rightarrow_{f_1} \cdots \rightarrow_{f_{k-1}} w_k = v\rangle$ lies completely in G, so all TTFs are exact and we can update the label of v by computing

$$\tau[v] := \min\{\tau[v], \mathbf{arr}\, f_{k-1}(\ldots \mathbf{arr}\, f_2(\mathbf{arr}\, f_1(\tau[u]))\ldots)\} .$$

Note that Algorithm 5.7 can be applied to expand shortcut edges in spite of the fact that no exact TTF is available. The reason is that the expansion procedure in Algorithm 5.7 requires the evaluation of TTFs only for original edges (see Line 11 to 14 of Algorithm 5.7). Checking the condition $\mathbf{arr}\, \underline{f}(q[u]) > r[v]$ before expanding a shortcut edge $u \rightarrow_{(\underline{f},\overline{f})} v$ prevents unnecessary edge expansions (see Line 9 of Algorithm 5.19).

Theorem 5.15. *Let H be an ATCH with relative error $\varepsilon > 0$. Then, with $s,t \in V$ and $\tau_0 \in \mathbb{R}$, Algorithm 5.15 returns an (s,t,τ_0)-EA-path in G.*

Proof. We show that $S^Y \subseteq H$ contains an (s,t,τ_0)-EA up-down-path in H, which implies that the expanded version of S^Y contains an (s,t,τ_0)-EA-path in G.

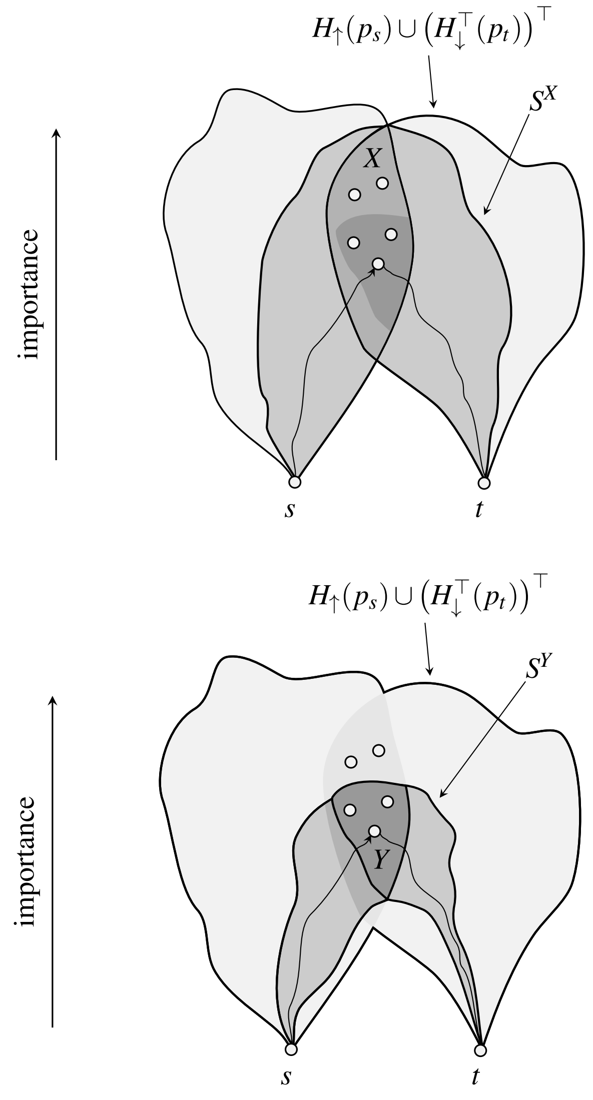

$H_{\uparrow}(p_s) \cup \left(H_{\downarrow}^{\top}(p_t)\right)^{\top}$
S^X
X
importance
s
t
$H_{\uparrow}(p_s) \cup \left(H_{\downarrow}^{\top}(p_t)\right)^{\top}$
S^Y
Y
importance
s
t

Figure 5.10. If the second phase (i.e., the thinning) of the ATCH-based EA query (see Algorithm 5.15) would be omitted, then the subgraph $S^X \subseteq H$ (top) would be extracted from H instead of the smaller subgraph $S^Y \subseteq H$ (bottom). In case of S^X the extraction would work by two BFSes that start from the candidate set X and that only process edges touched by forward and backward search, respectively, during the first phase (i.e., during bidirectional search). The second phase yields the supposedly smaller candidate set $Y \subseteq X$. So, the subgraph $S^Y \subseteq S^X \subseteq H$ contains supposedly less paths than S^X. Moreover, one of the downward BFSes that extract S^Y (namely, the one that yields $S^Y \cap H_{\downarrow}$) runs in the thinned graph $(H_{\downarrow}^{\top}(p_{\text{up}}))^{\top} \subseteq (H_{\downarrow}^{\top}(p_t))^{\top} \subseteq H_{\downarrow}$. This reduces $S^Y \cap H_{\downarrow}$ even more. If the subgraph S^Y is significantly smaller than the subgraph S^X, then the third phase (i.e., the corridor Dijkstra) runs significantly faster. Note that both S^X and S^Y contain an (s,t,τ_0)-EA up-down-path P_0, which enables the third phase to find an (s,t,τ_0)-EA path in G.

Algorithm 5.18. Instantiation of corridor Dijkstra, where time-dependent Dijkstra is performed on the fully expanded version $S \subseteq G$ of $S^Y \subseteq H$. The subprocedure *extractCone* (see Algorithm 5.11) is used to extract S^Y from the candidate set Y and the predecessor information p_s and p_{up} of forward and upward search respectively.

```
function $corridorDijkstraExpandFully(s,t : V,\ \tau_0 : \mathbb{R},\ p_s, Y, p_{\text{up}} : Reference) : Path$
    $S^Y := extractCone(s,Y,p_s) \cup \big(extractCone(t,Y,p_{\text{up}})\big)^\top$
                                   $// computes\ S_Y = \mathscr{P}_{H_\uparrow(p_s)}(s,Y) \cup \big(\mathscr{P}_{H_\downarrow^\top(p_{up})}(t,Y)\big)^\top$
    $S := S^Y$ with all shortcuts expanded for all departure times
    run time-dependent Dijkstra on graph $S$
    return $(s,t,\tau_0)$-EA path computed by time-dependent Dijkstra
```

Algorithm 5.19. Instantiation of corridor Dijkstra, where time-dependent Dijkstra is performed without expanding $S^Y \subseteq H$. Instead, shortcut edges $u \to v \in E_H \setminus E$ are expanded only when necessary (i.e., "on demand"). To do so, *expandPath* (see Algorithm 5.7) is invoked as subprocedure. The subprocedure *extractCone* (see Algorithm 5.11) is used to extract S^Y from the candidate set Y and the predecessor information p_s and p_{up} respectively.

```
function $corridorDijkstraEoD(s,t : V,\ \tau_0 : \mathbb{R},\ p_s, Y, p_{\text{up}} : Reference) : Path$
    $S^Y := (V_Y, E_Y) := extractCone(s,Y,p_s) \cup \big(extractCone(t,Y,p_{\text{up}})\big)^\top$
                                   $// computes\ S_Y = \mathscr{P}_{H_\uparrow(p_s)}(s,Y) \cup \big(\mathscr{P}_{H_\downarrow^\top(p_{up})}(t,Y)\big)^\top$
    $\tau[u] := \infty$ for all $u \in V_Y$, $\tau[s] := \tau_0$                        // initial node labels
    $p[u] := \bot$ for all $u \in V_Y$                              // initial predecessor information
    $Q := \{(s,\tau_0)\} : PriorityQueue$
    while $Q \neq \emptyset$ do
        $u := Q.deleteMin()$
        foreach $u \to_{(\underline{f},\overline{f})} v \in E_Y$ do
            if arr $\underline{f}(q[u]) > r[v]$ then continue
            $\tau_{\text{new}} :=$ arr $f_P(\tau[u])$ with $P := expandPath(u \to_{(\underline{f},\overline{f})} v)$
            if $\tau_{\text{new}} \geq \tau[v]$ then continue
            $\tau[v] := \tau_{\text{new}}$                                   // update node label of v
            $p[v] := u$                               // update predecessor information of v
            if $v \notin Q$ then $Q.insert(v,\ \tau[v])$
            else $Q.updateKey(v,\ \tau[v])$                                  // maintain PQ
```

Theorem 5.5 ensures the existence of a prefix-optimal (s,t,τ_0)-EA up-down-path with top node $x_0 \in X$ in H. So, Lemma 4.31 and 4.36 imply that S^X contains an EA up-down-path whose upward part is prefix optimal and whose downward part is an $(x_0,t,\mathrm{EA}_H(s,t,\tau_0))$-EA-path contained in

$$S^X_\downarrow := \big(\mathscr{P}_{H^\top_\downarrow(p_t)}(t,X)\big)^\top = S^X \cap H_\downarrow \; .$$

We argue similarly as in case of Theorem 5.6 and consider the graph G_0 consisting of $S^X_\downarrow$, the node s, and edges $s \to_f x$ with $f :\equiv \mathrm{TT}_{H_\uparrow}(s,x,\tau_0)$ for all $x \in X$. Every (s,t,τ_0)-EA-path in G_0 corresponds to an (s,t,τ_0)-EA up-down path in H. As lower and upper bound of each such TTF f we use $\underline{f} :\equiv q_s[x] - \tau_0$ and $\overline{f} :\equiv r_s[x] - \tau_0$ respectively. The first part of the second phase (i.e., the downward search) is obviously equivalent to an EA interval search on G_0, which implies

$$\begin{aligned}[q_{\mathrm{down}}[t], r_{\mathrm{down}}[t]] \ni \mathrm{EA}_{G_0}(s,t,\tau_0) &= \mathrm{EA}_{S^X_\downarrow}\big(x_0,t,\mathrm{EA}_H(s,x_0,\tau_0)\big)\\ &= \mathrm{EA}_H(x_0,t,\mathrm{EA}_H(s,x_0,\tau_0))\end{aligned}$$

and, w.l.o.g., that $H_\downarrow(p_{\mathrm{down}})$ contains an $(x_0,t,\mathrm{EA}_H(s,x_0,\tau_0))$-EA-path according to Lemma 4.31 (otherwise, an equivalent statement holds for another node $x'_0 \in X$).

The second part of the second phase (i.e., the upward search) is an LD interval search on the predecessor graph of the downward search; that is, on $H_\downarrow(p_{\mathrm{down}})$. Lemma 4.43 ensures that the transpose predecessor graph of the upward search,

$$\big(H^\top_\downarrow(p_{\mathrm{up}})\big)^\top \subseteq H_\downarrow(p_{\mathrm{down}}) \subseteq S^X_\downarrow \subseteq H_\downarrow \; ,$$

contains an (x_0,t,τ)-EA-path for all $\tau \in \mathrm{LD}_{H_\downarrow(p_{\mathrm{down}})}(x_0,t,[q_{\mathrm{down}}[t], r_{\mathrm{down}}[t]])$. Because of

$$\begin{aligned}&\mathrm{LD}_{H_\downarrow(p_{\mathrm{down}})}\big(x_0,t,[q_{\mathrm{down}}[t], r_{\mathrm{down}}[t]]\big)\\ &\quad\supseteq \mathrm{LD}_H\big(x_0,t,\mathrm{EA}_H\big(x_0,t,\mathrm{EA}_H(s,x_0,\tau_0)\big)\big)\\ &\quad= \mathrm{EA}^{-1}_H\big(x_0,t,\mathrm{EA}_H\big(x_0,t,\mathrm{EA}_H(s,x_0,\tau_0)\big)\big) \ni \mathrm{EA}_H(s,x_0,\tau_0) \; ,\end{aligned}$$

this includes an $(x_0,t,\mathrm{EA}_H(s,x_0,\tau_0))$-EA-path, which is also contained in

$$\big(\mathscr{P}_{H^\top_\downarrow(p_{\mathrm{up}})}(t,Y)\big)^\top = S^Y \cap H_\downarrow \; .$$

So, S^Y contains the desired (s,t,τ_0)-EA up-down-path. □

Min-Max-TCHs. Note that Algorithm 5.15 also works with Min-Max-TCHs. But in this case, the upward search does not have more information than the backward search at hand. So, the extra work needed to perform downward and upward Search does not pay off on this case. For this reason, we omit the second phase and set $S^Y := S^X$ for Min-Max-TCHs.

Stall-on-Demand. To make the ATCH-based EA query run even a little faster, we apply stall-on-demand to its first phase. Regarding the backward search, which is the backward version of TTP interval search, stalling and propagation work exactly in the same way as in case of the backward search of the TCH-based EA query (see Section 5.3.3). Regarding the forward search, which is an EA interval search, stalling and propagation work in a way that mixes the methods applied in case of time-dependent Dijkstra and TTP interval search. More precisely, by stalling and propagation we try to find paths

$$P_{wuv} := \langle w \rightarrow_{(\underline{f},\overline{f})} u \rightarrow_{(\underline{g_1},\overline{g_1})} \cdots \rightarrow_{(\underline{g_k},\overline{g_k})} v \rangle$$

with $\langle u \rightarrow_{(\underline{g_1},\overline{g_1})} \cdots \rightarrow_{(\underline{g_k},\overline{g_k})} v \rangle \subseteq H_\uparrow(p_s) \subseteq H_\uparrow$ and $w \rightarrow_{(\underline{f},\overline{f})} u \in E_\downarrow$, such that

$$q_s[v] > \mathbf{arr}\,\overline{g_k} \circ \cdots \circ \mathbf{arr}\,\overline{g_1} \circ \mathbf{arr}\,\overline{f}(r_s[w]) \tag{5.20}$$

is fulfilled. The existence of such a path P_{wuv} implies that the current $H_\uparrow(p_s)$ does not contain any path that is an (s,v,τ_0)-EA path with respect H, which means that the forward search can be pruned at v at that time. This is because Equation (5.20) together with Lemma 4.26 implies

$$\begin{aligned}\mathrm{EA}_{H_\uparrow(p_s)}(s,v,\tau_0) \geq q_s[v] &> \mathbf{arr}\,\overline{g_k} \circ \cdots \circ \mathbf{arr}\,\overline{g_1} \circ \mathbf{arr}\,\overline{f}(r_s[w]) \\ &\geq \mathbf{arr}\left(g_k * \cdots * g_1 * f * f_{\langle s \rightarrow \cdots \rightarrow w \rangle}\right)(\tau_0) \\ &\geq \mathrm{EA}_H(s,v,\tau_0)\end{aligned}$$

for some path $\langle s \rightarrow \cdots \rightarrow w \rangle \subseteq H_\uparrow(p_s) \subseteq H$. To detect paths like P_{wuv} that fulfill Equation (5.20) we maintain a value $r_{s,\mathrm{stall}}[v]$ for all nodes $v \in V$ and perform stalling and propagation analogously to Section 5.3.3.

5.4.3 Travel Time Profile Queries

An ATCH structure H with relative error ε can also be used to compute $\mathrm{TT}_G(s,t,\cdot)$ exactly (see Algorithm 5.20). The method partly applies similar ideas as in case of ATCH-based EA queries (see Section 5.4.2) and also runs in three phases. As usual, the different edge relaxations have been factored out to prevent too long pseudocode (see Algorithm 5.21 and Algorithm 5.22).

Phase 1: Bidirectional TTP Interval Search. In the first phase (see Lines 2 to 15), we perform a bidirectional TTP interval search to obtain an opening cone $S_\uparrow^X \subseteq H_\uparrow$ and a closing cone $S_\downarrow^X \subseteq H_\downarrow$. In principle, this is the same as the initial step performed by the accelerated version of the TCH-based TTP query (see Algorithm 5.10).

Algorithm 5.20. TTP query using an ATCH with $\varepsilon > 0$. After building the cones $S^X_\uparrow \subseteq H_\uparrow$ and $S^X_\downarrow \subseteq H_\downarrow$, we thin them out yielding $S^Y \subseteq S^X_\uparrow \cup S^X_\downarrow \subseteq H$. Expanding S^Y completely yields a corridor $S \subseteq G$ containing an (s,t,τ)-EA-path in G for all $\tau \in \mathbb{R}$. Contracting S yields $\mathrm{TT}_G(s,t,\cdot)$. As subroutines *extractCone* (see Algorithm 5.11), *approxTtpIntervalRelax* (see Algorithm 5.21), *approxTtpRelax* (see Algorithm 5.22), and *corridorContraction* (see Algorithm 5.23) are invoked.

```
function $atchTtpQuery(s,t : V, \tau_0 : \mathbb{R}) : TTF$
    $[q_s[u], r_s[u]] := [q_t[u], r_t[u]] := [\infty,\infty]$, $p_s[u] := p_t[u] := \emptyset$ for all $u \in V$
    $[q_s[s], r_s[s]] := [q_t[t], r_t[t]] := [0,0]$
    $B := \infty$
    $Q_s := \{(s,0)\}, Q_t := \{(t,0)\} : PriorityQueue$
    $X := \emptyset, \Delta := t$
    while $(Q_s \neq \emptyset$ or $Q_t \neq \emptyset)$ and $\min\{Q_s.min(), Q_t.min()\} \leq B$ do
        if $Q_{\neg\Delta} \neq \emptyset$ then $\Delta := \neg\Delta$            // with $\neg s := t$ and $\neg t := s$
        $u := Q_\Delta.deleteMin()$
        if $B < \infty$ and $q_s[u] + q_t[u] \leq B$ then $X := X \cup \{u\}$
        $B := \min\{B, r_s[u] + r_t[u]\}$
        for $u \to_{(\underline{f},\overline{f})} v \in E_\Delta$ do            // with $E_s := E_\uparrow$ and $E_t := E_\downarrow^\top$
            $approxTtpIntervalRelax\big(u \to_{(\underline{f},\overline{f})} v, q_\Delta, r_\Delta, p_\Delta, Q_\Delta\big)$
    if $X = \emptyset$ then return $\infty$
    $(S^X_\uparrow, S^X_\downarrow) := \big(extractCone(s,X,p_s), \big(extractCone(t,X,p_t)\big)^\top\big)$
                    // computes $S^X_\uparrow = \mathscr{P}_{H_\uparrow(p_s)}(s,X)$ and $S^X_\downarrow := \big(\mathscr{P}_{H_\downarrow^\top(p_t)}(t,X)\big)^\top$
    $\underline{F_s}[u] := \overline{F_s}[u] := \underline{F_t}[u] := \overline{F_t}[u] := \infty$, $p'_s[u] := p'_t[u] := \emptyset$ for all $u \in V$
    $\underline{F_s}[s] := \overline{F_s}[s] := \underline{F_t}[t] := \overline{F_t}[t] :\equiv 0$
    $Q'_s := \{(s,0)\}, Q'_t := \{(t,0)\} : PriorityQueue$
    $Y := \emptyset, \Delta := t$
    while $(Q'_s \neq \emptyset$ or $Q'_t \neq \emptyset)$ and $\min\{Q'_s.min(), Q'_t.min()\} \leq B$ do
        if $Q'_{\neg\Delta} \neq \emptyset$ then $\Delta := \neg\Delta$            // with $s := \neg t$, $t := \neg s$
        $u := Q'_\Delta.deleteMin()$
        if $\min \underline{F_s}[u] + \min \underline{F_t}[u] \leq B$ then $X := X \cup \{u\}$
        $B := \min\big\{B, \max \overline{F_s}[u] + \max \overline{F_t}[u]\big\}$
        for edge $u \to_{(\underline{f},\overline{f})} v$ in $S^X_\Delta$ do            // with $S^X_s := S^X_\uparrow$ and $S^X_t := S^X_\downarrow$
            if $\Delta = s$ then $approxTtpRelax\Big(u, v, \underline{f} * \underline{F_s}[u], \overline{f} * \overline{F_s}[u], \underline{F_s}, \overline{F_s}, p'_s, Q'_s\Big)$
            else $approxTtpRelax\Big(u, v, \underline{F_t}[u] * \underline{f}, \overline{F_t}[u] * \overline{f}, \underline{F_t}, \overline{F_t}, p'_t, Q'_t\Big)$
    $S^Y := extractCone(s,Y,p'_s) \cup \big(extractCone(t,Y,p'_t)\big)^\top$
                    // computes $S^Y = \mathscr{P}_{H_\uparrow(p'_s)}(s,Y) \cup \big(\mathscr{P}_{H_\downarrow^\top(p'_t)}(t,Y)\big)^\top$
    $S := S^Y$ with all shortcuts fully expanded for all departure times
    return $corridorContraction(s,t,S)$
```

Algorithm 5.21. Edge relaxation procedure as in TTP interval search (see Algorithm 4.3) but with lower and upper bounds $\underline{f}$ and $\overline{f}$, respectively, instead of an exact TTF, which may not be available in an ATCH. The *Reference* parameters q, r, p, Q are necessary to provide the context of the calling TTP interval search.

```
1 procedure approxTtpIntervalRelax($u \to_{(\underline{f},\overline{f})} v : Edge, q, r, p, Q : Reference$)
2     $[q_{\text{new}}, r_{\text{new}}] := [q[u] + \min \underline{f},\ r[u] + \max \overline{f}]$
3     if $q_{\text{new}} > r[v]$ then return
4     if $r_{\text{new}} < q[v]$ then $p[v] := \emptyset$
5     $p[v] := \{u\} \cup p[v]$
6     if $q_{\text{new}} \geq q[v]$ and $r_{\text{new}} \geq r[v]$ then return
7     $[q[v], r[v]] := [\min\{q[v], q_{\text{new}}\},\ \min\{r[v], r_{\text{new}}\}]$
8     if $v \notin Q$ then $Q.insert(v,\ q[v])$
9     else $Q.updateKey(v,\ q[v])$
```

Algorithm 5.22. Edge relaxation procedure as in approximate TTP search (see Algorithm 4.4). The *Reference* parameters q, r, p, Q are necessary to provide the context of the calling approximate TTP search.

```
1 procedure approxTtpRelax($u, v : V,\ \underline{g}_{\text{new}}, \overline{g}_{\text{new}} : TTF,\ q, r, p, Q : Reference$)
2     if $\underline{g}_{\text{new}}(\tau) \geq \overline{F}[v](\tau)$ for all $\tau \in \mathbb{R}$ then return
3     if $\overline{g}_{\text{new}}(\tau) < \underline{F}[v](\tau)$ for all $\tau \in \mathbb{R}$ then $p[v] := \emptyset$
4     $p[v] := \{u\} \cup p[v]$
5     if $\underline{g}_{\text{new}}(\tau) \geq \underline{F}[v](\tau) \wedge \overline{g}_{\text{new}}(\tau) \geq \overline{F}[v](\tau)$ for all $\tau \in \mathbb{R}$ then return
6     $(\underline{F}[v], \overline{F}[v]) := \Big(\min\big(\underline{F}[v],\ \underline{g}_{\text{new}}\big),\ \min\big(\overline{F}[v],\ \overline{g}_{\text{new}}\big)\Big)$
7     if $v \notin Q$ then $Q.insert(v,\ \min \underline{F}[v])$
8     else $Q.updateKey(v,\ \min \underline{F}[v])$
```

Phase 2: Bidirectional Approximate TTP Search. In the second phase (see Lines 16 to 28), we thin out the cones $S^X_\uparrow$ and $S^X_\downarrow$ by a bidirectional approximate TTP search. The forward search is an approximate TTP search (see Section 4.2.4) running on $H_\uparrow$, the backward search is the backward version of approximate TTP search (see Section 4.3.3) running on $H_\downarrow$. The idea of thinning out cones has already been applied in the context of the second phase of the ATCH-based EA query. However, we not only thin out the cone build by the backward search but also the cone build by the forward search here. This way, we obtain the subgraph $S^Y \subseteq S^X_\uparrow \cup S^X_\downarrow \subseteq H$, which contains an (s,t,τ)-EA up-down-path for all $\tau \in \mathbb{R}$. The subgraph S^Y is made up by the cones

$$S^Y_\uparrow := \mathscr{P}_{H_\uparrow(p'_s)}(s, Y) \subseteq S^X_\uparrow \subseteq H_\uparrow$$

Algorithm 5.23. Given a subgraph $S \subseteq G$, where s,t are nodes in S, this algorithm computes $\mathrm{TT}_S(s,t,\cdot)$ by contracting all nodes in S except for s and t. After termination, the only remaining edge is $s \rightarrow_f t$ fulfilling $f = \mathrm{TT}_S(s,t,\cdot)$. By contracting "easier" nodes earlier, the algorithm tries to keep the running time small.

```
function corridorContraction($s,t : V$, $S : Graph$) : TTF
    $Q := \emptyset$ : PriorityQueue
    foreach node $x$ in $S$ with $x \neq s,t$ do
        $c_x := \sum_{\langle u \rightarrow_f x \rightarrow_g v\rangle \subseteq S} |f| + |g|$      // estimate number of new bend points...
        $Q.insert(x, c_x)$                                  // ... and use the result as PQ key
    while $Q \neq \emptyset$ do
        $x := Q.deleteMin()$                   // contract the presumably easiest node next
        foreach path $\langle u \rightarrow_f x \rightarrow_g v\rangle \subseteq S$ do
            remove edges $u \rightarrow_f x$ and $x \rightarrow_g v$ from $S$
            if no edge $u \rightarrow_h v$ in $S$ then add $u \rightarrow_{g*f} v$ to $S$          // insert or...
            else replace $u \rightarrow_h v$ by $u \rightarrow_{\min(h,g*f)} v$ in $S$          // ...merge edge
        remove $x$ from $S$
    for all former neighbors $y$ of $x$ with $y \neq s,t$ do
        $c_y := \sum_{\langle u \rightarrow_f y \rightarrow_g v\rangle \subseteq S} |f| + |g|$              // recompute PQ key of y
        $Q.updateKey(y, c_y)$
    return $f$ with $S = \langle s \rightarrow_f t\rangle$          // at this point, S consists of only one edge
```

and

$$S_\downarrow^Y := \left(\mathscr{P}_{H_\downarrow^\top(p_t')}(t,Y)\right)^\top \subseteq S_\downarrow^X \subseteq H_\downarrow\,,$$

which we extract from predecessor information p_s' and p_t' of the second phase by simply running two BFSes starting from Y using Algorithm 5.11—just how the cones $S_\uparrow^X$ and $S_\downarrow^X$ are extracted from p_s and p_t.

Phase 3: Corridor Contraction. In the third phase (see Lines 29 and 30), we expand all the shortcuts in S^Y completely for all departure times (Line 29). This yields the relatively thin corridor $S \subseteq G$, which contains an (s,t,τ)-EA-path for all $\tau \in \mathbb{R}$. It would be possible to perform a TTP search on S, as all edges in S have exact TTFs. However, this would still take its time, even on a very thin corridor. Instead, we *contract* the whole *corridor* S (see Line 30 and Algorithm 5.23), which is much faster. Contracting a corridor means to contract one node of the corridor after another. More precisely, we remove one node of S after another while inserting shortcuts. This is very similar to the construction phase during TCH preprocessing (see Section 5.2.2), where we contract all the nodes of the road network G one after another. However, the contraction of a corridor is performed *online* and not *offline* as in case of preprocessing. For this reason, we do not have the time to check whether a shortcut is necessary or not. Instead, we simply insert *all* shortcuts.

During corridor contraction we use a PQ to control the order in that the nodes are contracted. As PQ key of a node $x \in V$ we use

$$c_x := \sum_{\langle u \to_f x \to_g v \rangle \subseteq S} |f| + |g| \; .$$

As we expect that $|g| + |f|$ and $|g * f|$ are quite correlated (see Corollary 3.11), we think c_x is a good estimate of the total complexity of the TTFs created by contracting x. But every bend point probably produces more bend points during repeated link and minimum operations. So, we always contract a node x with minimal c_x next, because this helps to keep the number of bend points low and this helps to reduce the running time of corridor contraction. This actually reveals why TTP search is so slow: There, nodes are processed in an order that does not pay attention to the number of newly created bend points, which is likely to increase the total number of processed bend points, and thus the running time, a lot.

Theorem 5.16. *Let H be an ATCH with relative error $\varepsilon > 0$. Then, for all $s,t \in V$ the ATCH-based TTP query (see Algorithm 5.20) returns* $\mathrm{TT}_G(s,t,\cdot)$.

Proof. We argue as in the correctness proof of the accelerated version of TCH-based TTP query (see Algorithm 5.10 in Section 5.3.2) and obtain that $S_\uparrow^X \cup S_\downarrow^X$ contains an (s,t,τ)-EA up-down-path with top node x_τ for all $\tau \in \mathbb{R}$ (see the proof of Theorem 5.9, where $S_\uparrow$ and $S_\downarrow$ play the roles of $S_\uparrow^X$ and $S_\downarrow^X$). The upward and the downward part of this up-down-path, respectively, lies completely in $S_\uparrow^X$ and $S_\downarrow^X$. So, both forward and backward search of the bidirectional approximate TTP search on $S_\uparrow^X \cup S_\downarrow^X$ reach x_τ. This implies $x_\tau \in X$, and with Lemma 4.23 and 4.39 we find that $H_\uparrow(p'_s) \cup (H_\downarrow^\top(p'_t))^\top$ also contains an (s,t,τ)-EA up-down-path with top node x_τ, say $R_\tau = \langle s \to \cdots \to x_\tau \to \cdots \to t \rangle$. So, R_τ is contained in S_Y. Thus, $S \subseteq G$ contains an (s,t,τ)-EA-path for all $\tau \in \mathbb{R}$. So, *corridorContraction* returns the sought-after TTP. □

Min-Max-TCHs. In case of Min-Max-TCHs we omit the second phase; that is, we do not thin out $S_\uparrow^X$ and $S_\downarrow^X$. Instead, we expand $S_\uparrow^X \cup S_\downarrow^X$ and perform the corridor contraction directly after the first phase; that is, after the bidirectional TTP interval search. This is because there is no information present in a Min-Max-TCH that can be used to further reduce the size of $S_\uparrow^X$ and $S_\downarrow^X$.

Stall-on-Demand. Section 5.3.3 explains how stall-on-demand can be used to further speed up the accelerated version of the TCH-based TTP query (see Algorithm 5.10 in Section 5.3.2). There, stall-on-demand is only applied to the first phase of the algorithm; that is, to the bidirectional TTP interval search. In the same way, stall-on-demand can be applied to the first phase of the ATCH-based TTP query, which is a bidirectional TTP interval search, too.

5.5 Inexact Space Efficient Querying

In practice, the accuracy of the input data may be arguable, or results with some error may simply be good enough. In such cases, we can use an *inexact TCH* structure (see Section 5.5.1). With inexact TCHs we can perform both inexact EA and inexact TTP queries. However, inexact TTP queries (see Section 5.5.2) are the kind of inexact queries we are actually interested in because ATCH-based EA queries, which we describe in Section 5.4.2, are already a space efficient method for fast and exact answering of EA queries. Inexact TTP queries, in contrast, provide an enormous further speedup compared to exact TTP queries. This justifies the loss of accuracy. Yet, the benefit of inexact EA queries (see Section 5.5.3) is that we can use the same hierarchical representation of the road network for both EA and TTP queries; namely, an inexact TCH structure. Otherwise, we had to keep both an ATCH structure and an inexact TCH structure in memory at the same time. This might already cost too much space.

5.5.1 Inexact Time-Dependent Contraction Hierarchies

An *inexact TCH* structure with *relative error* $\varepsilon > 0$ is generated from an exact TCH structure H by replacing *every* TTF f by an *inexact TTF* $\tilde{f}$ that fulfills

$$(1+\varepsilon)^{-1} f(\tau) \leq \tilde{f}(\tau) \leq (1+\varepsilon) f(\tau) \tag{5.21}$$

for all $\tau \in \mathbb{R}$. To compute $\tilde{f}$ from f we use Neubauer's [66] implementation of the Imai-Iri algorithm [52] and restore the FIFO property if necessary. This is similar to the computation of approximate TTFs as they occur in the context of ATCHs (see Section 5.4.1). Recall that the Imai-Iri algorithm computes an approximate TTF of minimal possible complexity for a given relative error $\varepsilon > 0$. Note that we not only replace the TTFs of the shortcut edges—as we do in case of ATCHs—but also the TTFs of the original edges (i.e., of all edges in E_H including $E \subseteq E_H$). Also, we annotate every edge $u \rightarrow_{\tilde{f}} v$ in an inexact TCH with the *conservative bounds*

$$\min\{\min f, \min \tilde{f}\} \quad \text{and} \quad \max\{\max f, \max \tilde{f}\} \ .$$

Note that the conservative bounds are not needed in the context of ATCHs, because $\min \underline{f}$ and $\max \overline{f}$, respectively, are already a lower and an upper bound of f. For $\min \tilde{f}$ and $\max \tilde{f}$, in contrast, this is not the case. Inexact TCHs have similar memory usage as ATCHs and can be viewed as space efficient in compared to exact TCHs hence.

It is also important to note that the relative error of the computed inexact results is *not* limited by ε. In theory, the relative error can even get quite large. As a consequence, our correctness proofs for inexact EA and TTP querying (see Theorem 5.17 and 5.19) only show that these algorithms return *some* path (or *some* TTF respectively) if the destination is reachable from the start. But in our experiments, we only observe small errors (see Section 5.6). Note that Geisberger and Sanders [42] give an upper bound for the relative error depending on the maximum slopes of TTFs. However, we do not exploit this error bound in this thesis.

5.5.2 Inexact Travel Time Profile Queries

With inexact TCHs, we only get an approximation of $\mathrm{TT}_G(s,t,\cdot)$. But compared to the ATCH-based TTP query (see Section 5.4.3), which is our fastest algorithm for exact TTP queries, we get an enormous speedup (see Section 5.6). To achieve this performance, we simply apply the accelerated version of the TCH-based TTP query (see Algorithm 5.10 in Section 5.3.2) to inexact TCH structures. Only the first phase (i.e., the bidirectional TTP interval search) and the stall-on-demand, which is applied during this phase, require a small modification: In both cases the *conservative* bounds must be used instead of the bounds $\min \tilde{f}$ and $\max \tilde{f}$. Otherwise, it can happen that stall-on-demand stalls so many nodes that forward and backward search do not meet—even if t *is* reachable from s. Then, the query would return ∞, which means that the algorithm did not find any path from s to t.

Theorem 5.17. *The inexact TCH-based TTP query returns an inexact TTP other than ∞ if, and only if, the destination node t is reachable from start node s.*

Proof. Consider an inexact TCH structure $\tilde{H}$ with relative error $\varepsilon > 0$. If t is not reachable from s in G, then $\tilde{H}$ does not contain a path from s to t, too. Otherwise, $\tilde{H}$ contains one or more up-down-paths from s to t. This is due to the guaranteed existence of prefix optimal EA up-down-paths in the corresponding exact TCH structure H (see Theorem 5.5). This and the fact that we use a modified version of Algorithm 5.10 that works in terms of the conservative bounds implies that $S_\uparrow \cup S_\downarrow$ also contains an up-down-path from s to t after the bidirectional TTP interval search is finished (see the proof of Lemma 5.9). But this means that the forward and the backward search of Algorithm 5.8 invoked by Algorithm 5.10 surely meet in one or more candidate nodes. The returned TTF is different from ∞ hence. □

Stall-on-Demand. Like in case of exact TTP queries, stall-on-demand can also be used to make inexact TTP queries faster. As the query algorithm is actually Algorithm 5.10, where the first phase uses the conservative bounds, stall-on-demand works as described in Section 5.3.3 but with conservative bounds.

It is important to note, however, that stall-on-demand may change the computed inexact TTP. An inexact TTP for traveling from s to t, which is computed with stall-on-demand, may be different from an inexact TTP for traveling from s to t computed without stall-on-demand. Note that stall-on-demand can only increase the resulting inexact TTP. But whether a TTP computed with stall-on-demand is less accurate than without is unclear. We can only guarantee that the inexact TTP query computes *some* TTP different from ∞, both with or without stall-on-demand.

Theorem 5.18. *The inexact TCH-based TTP query returns an inexact TTP other than ∞ if, and only if, the destination node t is reachable from the start node s, even if stall-on-demand is applied (though the resulting TTP may be increased).*

Proof. We only give a prove for the forward search. For the backward search one can argue analogously.

Like in the proof of Theorem 5.17 we consider an inexact TCH structure $\tilde{H}$ with relative error $\varepsilon > 0$. Due Theorem 5.5 the corresponding exact TCH structure H contains a prefix-optimal (s,t,τ)-EA up-down-path with top node x_τ for all $\tau \in \mathbb{R}$ if, and only if, t is reachable from s in the original road network G. Consider stall-on-demand with respect to the forward search. That the search is pruned at a node v implies that

$$q_s[v] > r_s[w] + \max\big\{\max f, \max \tilde{f}\big\} + \sum_{i=1}^{k} \max\big\{\max g_i, \max \tilde{g}_i\big\}$$

is fulfilled with $w \to_f u \in E_\downarrow$ and $\langle u \to_{g_1} \cdots \to_{g_k} v\rangle \subseteq H_\uparrow$ at the time when the node u is removed from the PQ. This means, that a path

$$P_{swuv} := \big\langle s \to \cdots \to w \to_f u \to_{g_1} \cdots \to_{g_k} v \big\rangle$$

exists with $\langle s \to \cdots \to w\rangle \subseteq H_\uparrow(p_s)$, $w \to_f u \in E_\downarrow$, and $\langle u \to_{g_1} \cdots \to_{g_k} v\rangle \subseteq H_\uparrow$, such that

$$\begin{aligned}
\mathrm{TT}_{H_\uparrow}(s,v,\tau) &\geq q_s[v] \\
&> r_s[w] + \max\{\max f, \max \tilde{f}\} + \textstyle\sum_{i=1}^{k} \max\{\max g_i, \max \tilde{g}_i\} \\
&\geq r_s[w] + \max f + \max g_1 + \cdots + \max g_k \\
&\geq f_{P_{suwv}}(\tau) \\
&\geq \mathrm{TT}_H(s,v,\tau)
\end{aligned}$$

holds for all $\tau \in \mathbb{R}$ with respect to Lemma 4.19. This means that the current $H_{\uparrow}(p_s)$ does not contain an (s,t,τ)-EA-path for any $\tau \in \mathbb{R}$ and the search can be pruned at v at that time. So, stall-on-demand based on conservative bounds prevents no paths that are prefix-optimal EA paths in $H_{\uparrow}$ with respect to the exact TCH structure H. But then, there is always an up-down-path left in $S_{\uparrow} \cup S_{\downarrow}$ and the algorithm returns an inexact TTP different from ∞ hence. □

5.5.3 Inexact Earliest Arrival Queries

For inexact EA queries, we perform Algorithm 5.24 on an inexact TCH structure. It is actually a variant of the TCH-based EA query specified in Algorithm 5.4. The difference regards the bidirectional phase, where not only the backward search but also the forward search is a TTP interval search. However, the downward search requires arrival times for the candidate nodes instead of travel time intervals. So, we have to perform an additional time-dependent Dijkstra search on $S_{\uparrow}$ (i.e., an *upward search*), before the downward search can be performed.

The inexact EA query algorithm specified in Algorithm 5.24 yields exact EA times when applied to an exact TCH structure. For inexact TCH structures, it returns an up-down-path from the start s to the destination t if, and only if, t is reachable from s. Note that we perform stall-on-demand during the bidirectional search, but only in terms of conservative bounds—just like in case of inexact TTP queries (see Section 5.5.2). Otherwise, we may again stall too many nodes, and no path from s to t may be found even if there is one. In fact, the only reason for performing a bidirectional TTP search as first phase is to make stall-on-demand applicable to EA queries in the presence of inexact data.

Theorem 5.19. *The inexact TCH-based EA query computes a (not necessary optimal) up-down-path if, and only if, t is reachable from s in G. For $\varepsilon = 0$ (i.e., the TCH is exact) it computes an (s,t,τ_0)-EA up-down-path.*

Proof. If t is not reachable from s, then there is no path from s to t in $\tilde{H}$. Otherwise, we argue as in the proof of Theorem 5.17 and find $S_{\uparrow} \cup S_{\downarrow}$, with

$$S_{\downarrow} := \left(\mathscr{P}_{H_{\downarrow}^{\top}(p_t)}(t,X)\right)^{\top} \subseteq H_{\downarrow}\,,$$

contains an up-down-path with top node x_0. So, the candidate set Y is surely not empty after the upward search, and the downward search reaches t hence. Thus, an arrival time other than ∞ and an up-down-path from s to t is returned.

For $\varepsilon = 0$ we argue that x_0 is reached by the upward search with final label $\mathrm{EA}_G(s,x_0,\tau_0)$, because $H_{\uparrow}(p_s)$ contains such a path due to Lemma 4.14. We further argue as in the correctness proof of the TCH-based EA query (see Theorem 5.6) that the downward search reaches t with final label $\mathrm{EA}_G(s,t,\tau_0)$. □

Algorithm 5.24. A variant of TCH-based EA query (Algorithm 5.4) using similar methods as Algorithm 5.10. For exact TCHs it returns the exact EA time as well as a corresponding EA up-down-path. For inexact TCHs it yields an inexact EA time as well as a not necessarily optimal up-down-path. As subroutines it invokes *tdRelax* (see Algorithm 5.5) and *ttpIntervalRelax* (see Algorithm 5.6).

```
function $tchConeEaQuery(s,t : V,\ \tau_0 : \mathbb{R}) : Path$
    $[q_s[u], r_s[u]] := [q_t[u], r_t[u]] := [\infty,\infty],\ p_s[u] := p_t[u] := \emptyset$ for all $u \in V$
    $[q_s[s], r_s[s]] := [q_t[t], r_t[t]] := [0,0]$
    $B := \infty$
    $X := \emptyset : Set$
    $Q_s := \{(s,0)\}, Q_t := \{(t,0)\} : PriorityQueue$
    $\Delta := t$
    while $(Q_s \neq \emptyset$ or $Q_t \neq \emptyset)$ and $\min\{Q_s.min(),\ Q_t.min()\} \leq B$ do
        if $Q_{\neg\Delta} \neq \emptyset$ then $\Delta := \neg\Delta$                    // with $\neg s := t$ and $\neg t := s$
        $u := Q_\Delta.deleteMin()$
        if $B < \infty$ and $q_s[u] + q_t[u] \leq B$ then $X := X \cup \{u\}$
        $B := \min\{B,\ r_s[u] + r_t[u]\}$
        for $u \to_f v \in E_\Delta$ do                    // with $E_s := E_\uparrow$ and $E_t := E_\downarrow^\top$
            $ttpIntervalRelax(u \to_f v, q_\Delta, r_\Delta, p_\Delta, Q_\Delta)$
    $S_\uparrow := extractCone(s, X, p_s)$                    // computes $S_\uparrow = \mathscr{P}_{H_\uparrow(p_s)}(s,X)$
    $\tau_{\text{up}}[u] := \infty,\ p_{\text{up}}[u] := \perp$ for all $u \in V,\ \tau_{\text{up}}[s] := \tau_0$
    $Q_{\text{up}} := \{(s, \tau_0)\} : PriorityQueue$                    // upward search
    $Y := \emptyset : Set$                    // candidate set
    while $Q_{\text{up}} \neq \emptyset$ do
        $u := Q_{\text{up}}.deleteMin()$
        if $\tau_{\text{up}}[u] > B + \tau_0$ then break
                    // B is upper bound of travel time not of arrival time
        if $\tau_{\text{up}}[u] + q_t[u] \leq B + \tau_0$ then $Y := Y \cup \{u\}$
        for $u \to_f v$ in $S_\uparrow$ do $tdRelax(u \to_f v, \tau_{\text{up}}, p_{\text{up}}, Q_{\text{up}})$
    $\tau_{\text{down}}[u] := \infty$ for all $u \in V$
    $Q_{\text{down}} := \emptyset : PriorityQueue$                    // PQ for downward search
    foreach $u \in Y$ do
        if $\tau[u] + q[u] \leq B$ then
            $\tau_{\text{down}}[u] := \tau[u],\ Q_{\text{down}}.insert(u, \tau[u])$
    while $Q_{down} \neq \emptyset$ do
        $u := Q_{\text{down}}.deleteMin()$
        if $u = t$ then return $\langle s = p_s[\dots p_s[t] \dots] \to \cdots \to p_s[t] \to t \rangle$
        for $v \in p_t[u]$ do $tdRelax(u \to_f v, \tau_{\text{down}}, p_s, Q_{\text{down}})$
```

5.6 Experimental Evaluation

All techniques described in this chapter are evaluated experimentally with respect to preprocessing time, query time, memory usage, and relative error in case of inexact techniques. The underlying test instances are road networks of Western Europe and Germany (see Section 5.6.1). After explaining the experimental setup (see Section 5.6.2) we present and discuss the observed results for preprocessing and querying (see Section 5.6.3). Finally, we compare the results with the performance of some goal-directed time-dependent route planning techniques (see Section 5.6.4).

5.6.1 Input Road Networks

As inputs we use two road networks, of Germany and Western Europe, both provided by PTV AG [71] for scientific use (see Table 5.1). For *Germany*, which has about 4.7 million nodes and about 10.8 million edges, we have five sets of time-dependent edge weights (i.e., TTFs) collected from historical data: They reflect the traffic of *Monday*, *midweek* (Tuesday till Thursday), *Friday*, *Saturday*, and *Sunday* each and have different percentage of time-dependent (i.e., non-constant) TTFs (see Table 5.2). For *Saturday* and *Sunday*, for example, there are fewer non-constant TTFs with less bend points (i.e., lower complexity). This seems to be natural as we expect less traffic at the weekend than during the week. We often use the abbreviations *Mon*, *mid*, *Fri*, *Sat*, and *Sun*.

For (Western) *Western Europe*, which has about 18 million nodes and about 42.2 million edges, we have two sets of edge TTFs. In both sets all non-constant TTFs are synthetically generated as described by Nannicini et al. [64]. The first set reflects a *medium (med)* amount of traffic where motorways and national roads can be congested; that is, only motorways and national roads have non-constant TTFs. Urban streets, local streets, and rural roads are not time-dependent; that is, they have constant TTFs assigned. The second set reflects a *high* amount of traffic. There, not only motorways and national roads but also urban roads have non-constant TTFs [20].

road network	$\|V\|$	$\|E\|$	out-degree
Germany	4.7 mio	10.8 mio	2.30
Western Europe	18.0 mio	42.2 mio	2.34

Table 5.1. Number of nodes ("$|V|$") and edges ("$|E|$") of the German and the Western European road network, which are the underlying test instances of the experiments. The average out-degree, which equals $|E|/|V|$, is also reported.

			avg. relative delay		complexity		
TTF set	space [B/n]	non-const. TTFs [%]	including const. [%]	excluding const. [%]	avg. incl. const. [#]	avg. excl. const. [#]	max. [#]
Germany							
Monday	96	7.0	2.5	36.1	2.1	16.8	52
midweek	95	7.2	2.6	36.0	2.1	16.3	51
Friday	88	6.3	2.2	34.9	1.9	15.8	53
Saturday	65	3.8	1.2	31.7	1.4	12.7	35
Sunday	55	2.4	0.7	29.1	1.3	11.8	37
Western Europe							
high	76	6.2	7.7	124.0	1.6	11.2	14
medium	47	1.0	1.2	123.5	1.1	11.2	14

Table 5.2. Some properties of our input road networks and the available TTF sets: the space usage in bytes per node ("B/n"), the percentage of time-dependent edge weights ("non-const. TTFs"), as well as the average ("avg.") relative delay and complexity of the TTFs, both including ("incl.") and excluding ("excl.") the constant ("const.") ones. The maximum ("max.") complexity that occurs in each TTF set is also reported.

Table 5.2 also reports the *average relative delay* of all sets of time-dependent edge weights; that is, of all sets of TTFs. We define *relative delay* of a TTF f as $(\max f - \min f)/\min f$. Table 5.2 reports the average relative delay over the TTFs of all edges, both including and excluding the constant ones. These are the values

$$delay_{\text{incl}} := \frac{1}{|E|} \cdot \sum_{u \to_f v \in E} \frac{\max f - \min f}{\min f}$$

and

$$delay_{\text{excl}} := \frac{1}{|E_{\text{excl}}|} \cdot \sum_{u \to_f v \in E_{\text{excl}}} \frac{\max f - \min f}{\min f} \,,$$

respectively, where E_{excl} is defined by $E_{\text{excl}} := \{u \to_f v \in E \mid f \text{ not constant}\}$. Note that Delling (e.g., see [21]) also reports an average relative delay, but over EA paths, not over edges. We think the relative delay is a good indicator of how well pruning in the context of TTP interval search works. Recall that TTP interval search uses travel time intervals as node labels that are computed on basis of $\min f$ and $\max f$ for TTFs f. So, larger delays result in wider intervals, which means that intervals are more likely to overlap. As a consequence, predecessor sets are less often emptied (see Line 10 of Algorithm 4.3).

Also, stall-on-demand (see Section 5.3.3) is more likely to fail for wider intervals when applied to TTP interval search. As all our query algorithms make use of TTP interval search, their running time should increase with the relative delay of the TTF sets. This also applies to the preprocessing, which utilizes TTP interval search to speed up witness search (see Section 5.2.2 on page 193). Note that a higher percentage of non-constant TTFs and a higher complexity of these TTFs also influences the running times, especially when TTP search is involved. This applies to the preprocessing as well as TTP queries. Our experiments show that a higher relative delay together with a higher percentage of non-constant TTFs and a higher complexity really increases the running times.

5.6.2 Setup

The experimental evaluation is done on a machine with two Core i7 Quad-Cores clocked at 2.67 Ghz with 48 GiB of RAM. The preprocessing is evaluated under SUSE Linux 11.1. with all programs compiled by GCC 4.3.2 at optimization level 3. The query performance is evaluated on the same machine, but under Ubuntu 12.04.2 with GCC 4.8.2, also at optimization level 3. To measure preprocessing time, we simply use the system timer. To measure query times, we count the number of CPU cycles passing by from the beginning to the end of every single query, multiplying the result with the CPU clock rate. Query times are always measured using one single thread if not stated otherwise.

All figures refer to the scenario that only the EA times and the TTPs have to be determined, without expanding the up-down-paths and outputting complete route descriptions. But when reporting memory consumption, we include the space needed to allow up-down-path expansion. Memory usage is often given in terms of the *total* space usage in average bytes per node. Table 5.2 reports the memory usage of all input road networks as original road networks; that is, the memory usage of the graph data structures used for time-dependent Dijkstra. For TCHs, ATCHs, Min-Max-TCHs, and inexact TCHs we also report the memory *overhead* compared to the original road network; sometimes as a *growth factor* and sometimes as an absolute value in bytes per node.

We measure the average running time of the different EA query algorithms by performing a bulk of 1 000 queries each. Therein, all the triples (s, t, τ_0) of start node s, destination node t, and departure time τ_0 are selected randomly from $V \times V \times [0\text{h}, 24\text{h})$. For TTP queries a departure time is not necessary of course. Note that we always execute such a bulk of 1 000 queries three times recording every single query time separately. Afterwards, for each of the 1 000 queries, we take the median of the three available query times and report the average of these 1 000 median running times as result. We do this to prevent accidental outliers which we observed in the past. Occasional activities triggered by the operating system during our experiments may be a possible source. For EA queries we not only report absolute average running times. We also report speedups with respect to the average running time of the one-to-one-version of time-dependent Dijkstra on the original road network. For TTP queries we are not able to report speedups, because running a plain TTP with 1 000 randomly selected pairs of start and destination nodes would be a matter of days or weeks.

To measure the errors, we use many more test cases: 1 000 000 random EA queries and 10 000 random TTP queries, where the error of TTP queries is measured for 100 random departure times each. We always report the average and the maximum relative error. Note that error is always determined with respect to the result of a time-dependent Dijkstra search in the original road network.

We also measure the machine-independent behavior of our algorithms: In all cases we count the number of PQ removals (i.e., invocations of *deleteMin*) and of touched edges. For time-dependent Dijkstra the number of touched edges is identical to the number of relaxed edges. For other query algorithms this also includes the number of edges copied by BFSes, of expanded shortcuts, and of edges processed during corridor contraction (as, e.g., in Algorithm 5.20). For EA queries we additionally count how often non-constant TTFs are evaluated (including similar operations like, e.g., computing $\max \mathbf{dep} \underline{f}(r[u])$ as in Algorithm 5.17 invoked by Algorithm 5.15). For TTP queries, in contrast, we count the number of bend points of the TTFs processed by link and minimum operations. In case of EA queries, we not only report the absolute numbers of PQ removals, touched edges, and processed bend points, but also the corresponding speedups compared to time-dependent Dijkstra.

5.6.3 Results

This section presents the results of our experimental evaluation of TCHs for the case that edge costs are time-dependent travel times. This includes TCH preprocessing (see below), EA and TTP queries (see page 253), as well as the reuse of node orders for a different TTF set (see page 262).

Preprocessing. First in our experimental evaluation, we consider the performance of the preprocessing; that is, of node ordering and TCH construction. There, we not only report the running times for sequential and parallel execution. We also evaluate the optimized witness search to find out how much the different optimizations reduce the running time. Moreover, we give some insight how the preprocessing behaves during its execution.

Sequential and Parallel Preprocessing Times. Table 5.3 shows the running times of node ordering (see Section 5.2.3) and TCH construction (see Section 5.2.2) for both *Germany* and *Western Europe*, each with all available TTF sets. The node ordering takes considerably longer than the construction. This is not surprising because the node ordering performs simulated contractions, which the TCH construction does not. The preprocessing takes longer the more non-constant TTFs are present in a TTF set, the higher the complexity of these TTFs is, and the larger the average relative delay is. The former two slow down TTP search. The latter makes it more likely that travel time intervals overlap during TTP interval search. So, TTP interval search may more often fail to prove that a shortcut is not needed.

Table 5.3 not only reports sequential preprocessing times (1 thread), but also parallel preprocessing times for shared memory (2, 4, and 8 threads). For both node ordering and construction, the parallelization scales pretty well. For the node ordering of Germany with TTFs reflecting Saturday traffic and 2 threads, we even observe a superlinear speedup. That the machine used in our experiments has a NUMA architecture is a plausible explanation. This means the data structures may partly get located outside the local memory of the involved CPU cores—and

		order		construct		order		construct	
road network	TTF set	time [h:m:s]	parl. spd.	time [h:m:s]	parl. spd.	time [h:m:s]	parl. spd.	time [h:m:s]	parl. spd.
		1 thread				2 threads			
Germany	Mon	0:28:23	-	0:07:41	-	0:14:42	1.9	0:03:54	2.0
	mid	0:29:21	-	0:07:33	-	0:14:57	2.0	0:03:57	1.9
	Fri	0:24:37	-	0:06:19	-	0:12:21	2.0	0:03:16	1.9
	Sat	0:15:09	-	0:03:45	-	0:07:23	2.1	0:02:00	1.9
	Sun	0:12:30	-	0:03:10	-	0:06:08	2.0	0:01:41	1.9
Western Europe	high	3:52:49	-	0:51:58	-	1:56:44	2.0	0:26:33	2.0
	med	1:31:37	-	0:21:42	-	0:48:29	1.9	0:11:51	1.8
		4 threads				8 threads			
Germany	Mon	0:07:53	3.6	0:02:06	3.7	0:04:50	5.9	0:01:16	6.1
	mid	0:08:06	3.6	0:02:09	3.5	0:05:04	5.8	0:01:14	6.1
	Fri	0:06:39	3.7	0:01:47	3.5	0:04:03	6.1	0:01:02	6.1
	Sat	0:04:01	3.8	0:01:08	3.3	0:02:33	5.9	0:00:41	5.5
	Sun	0:03:14	3.9	0:00:56	3.4	0:02:06	5.9	0:00:33	5.7
Western Europe	high	1:02:35	3.7	0:14:04	3.7	0:37:42	6.2	0:08:02	6.5
	med	0:27:42	3.3	0:06:14	3.5	0:17:49	5.1	0:03:40	5.9

Table 5.3. Running times of node ordering and TCH construction for all our road networks and TTF sets executed with 1, 2, 4, and 8 threads. For 2, 4, and 8 threads we also report the speedups achieved by parallel execution (“parl. spd.”).

this can be different for different runs of the preprocessing. This is likely to make the preprocessing times measured on this machine a little unstable.

Optimized Witness Search. The reader may remember that preprocessing mainly consists of many node contractions where we try to omit as many shortcuts as possible (see Section 5.2). There, we use some techniques to optimize the running time of the witness search, whose most basic form is an expensive TTP search. These techniques are: (i) a preceding sample search, (ii) a preceding TTP interval search with restricting the TTP search to a corridor in the predecessor graph of the TTP interval search, (iii) a heuristic to further thin out this corridor, and (iv) the caching of results of witness searches; with (i)–(iii) described in Section 5.2.2, and (iv) described in Section 5.2.3. We demonstrate the impact of (i)–(iii) on the running time of the node ordering by incrementally deactivating them. We do this both with (iv) activated and deactivated, both for sequential (1 thread) and parallel (8 threads) execution. Table 5.4 shows the resulting running times with a timeout of twelve hours. The impact of (i) (i.e., of sample search) seems to be rather small, though it can be noticed. The impact of (ii)–(iv) (i.e., TTP interval search and restricting to a corridor, heuristically thinning out this corridor, and caching of witnesses) is much larger. We can even say that all the techniques except sample search are vital for feasible preprocessing.

We also tested how preprocessing works without the *hop limit* described in Section 5.2.2. It turned out that the hop limit has little effect on the node ordering

		configuration			
	sample search	+	-	-	-
	heuristic thinning	+	+	-	-
	TTP interval search	+	+	+	-
		order time			
# threads	witness cache	[h:m:s]	[h:m:s]	[h:m:s]	[h:m:s]
1	+	0:29:21	0:35:25	4:36:34	$\geq$ 12 h
	-	1:37:57	1:52:32	$\geq$ 12 h	$\geq$ 12 h
8	+	0:05:04	0:05:45	1:16:19	$\geq$ 12 h
	-	0:15:53	0:17:17	4:00:17	$\geq$ 12 h

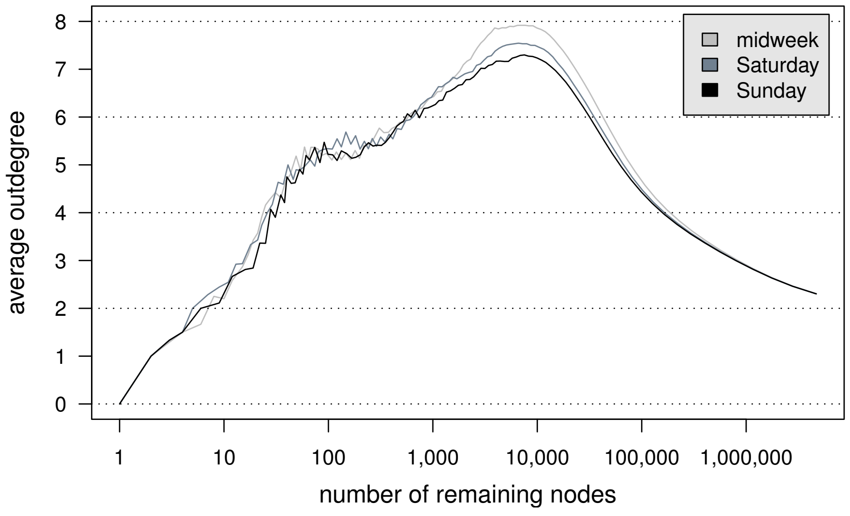
Table 5.4. Running time of node ordering for *Germany midweek* when the different techniques used to optimize the running time of the node ordering are successively deactivated ("+" means activated and "-" means deactivated). We report the running times of the resulting configurations for both sequential (1 thread) and parallel (8 threads) execution.

for *Germany midweek*, but for *Western Europe high* the node ordering gets infeasible without hop limit. So, a hop limit of 16 is always active. In the past we also used *aggressive edge reduction* [5]: During preprocessing, we periodically check for unnecessary shortcuts $u \rightarrow_f v$. In principle, we do this by a TTP search from u to v. Of course, this can be accelerated in a similar way as the witness search. However, in the current setting the edge reduction did not have much impact, so it was omitted.

During Preprocessing. Figure 5.11 and 5.12 give some insight into the TCH construction for the German road network with different sets of TTFs. It turns out that, for TTF sets with more non-constant TTFs, the number of inserted shortcuts does not increase too much. Still, we have a significantly higher complexity of the TTFs of these shortcuts. The majority of the preprocessing time is needed for the last 100 000 nodes. From around the last 1 000 nodes the construction and the node ordering get faster again. The higher relative delay is likely to raise the number of necessary TTP searches during witness search because the TTP interval search may more often fail to prove that a shortcut is not needed. The increased complexity of TTFs is also a probable source of longer running time because more complex TTFs mean that TTP search has to process more bend points, which makes TTP search slower. However, we have not examined that in detail.

Querying. We not only evaluate the performance of preprocessing but also of EA and TTP queries. TCH-based querying is, in contrast to the preprocessing, not parallelized because it already runs very fast. To measure as stable running times as possible, we force the CPU core that performs the query to put the data structures into its local memory, getting rid of NUMA effects this way.

Table 5.5 shows the running times of exact EA queries using TCHs. Table 5.6 shows the running times of exact EA and TTP queries using ATCHs with $\varepsilon = 2.5\,\%$ (this value seems to be a good compromise between space usage and running time). As expected, the TCH-based EA query (see Algorithm 5.4) is faster than the ATCH-based EA query (see Algorithm 5.15), but the necessary TCH structure needs much more space than the ATCH structure of course. The running times of EA queries for *Western Europe*, which has much more nodes and edges than *Germany*, suggest that these algorithms scale well with the size of the road network. The ATCH-based TTP query (see Algorithm 5.20) performs quite well as it shows running times considerably less than 0.1 sec for *Germany*. That means TTP queries can be answered "instantaneously", at least from the point of view of a human user. For *Western Europe* we can not provide instantaneous TTP queries, but the running times are still not bad. Altogether, it is not surprising that



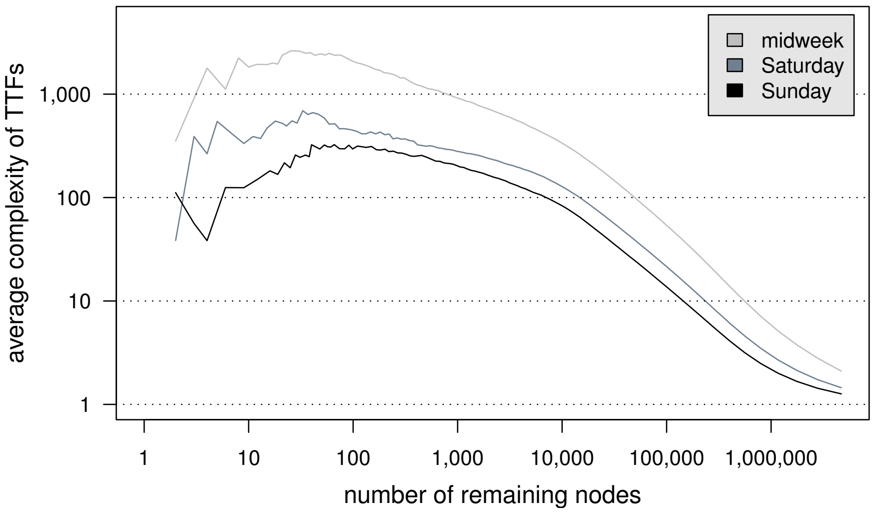
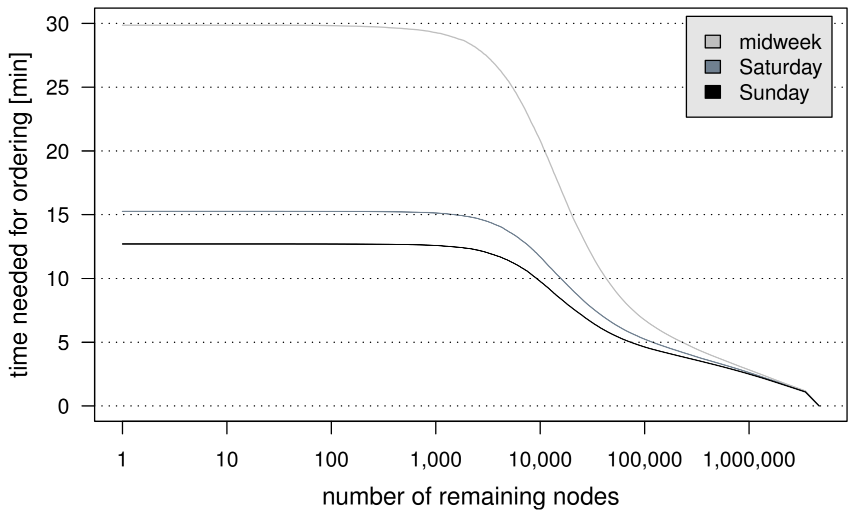


Figure 5.11. How average degree (top) and average TTF complexity (bottom) evolve in the remaining graph during TCH construction for *Germany midweek*, *Saturday*, and *Sunday*. As the x-axis shows the number of nodes not yet contracted, time “flows” from right to left in the above charts.



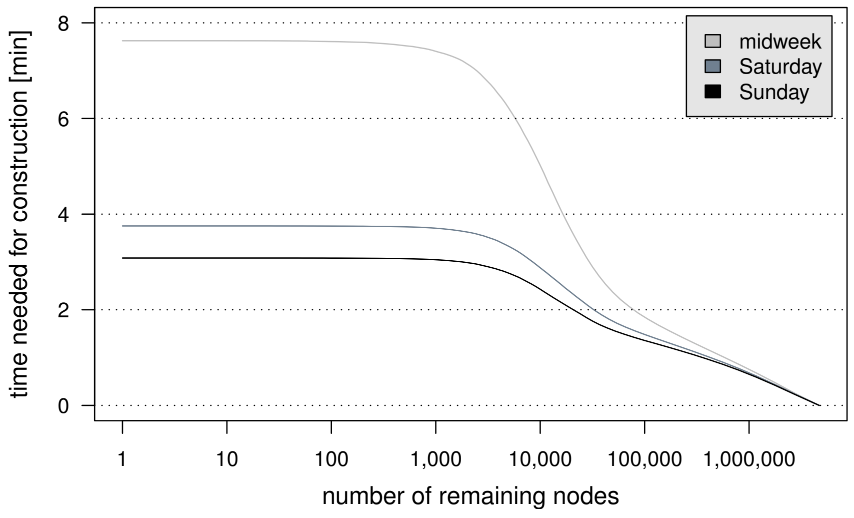


Figure 5.12. How the running time proceeds during node ordering (top) and TCH construction (bottom) for *Germany midweek*, *Saturday*, and *Sunday* using one single thread of execution. As the x-axis shows the number of nodes not yet contracted, time "flows" from right to left in the above charts.

TTP queries run faster the less time-dependency (i.e., less non-consntant TTFs, less bend points, and less average relative delay) a graph has.

As explained in Section 5.3.1 we apply time-dependent bidirectional search to make hierarchical routing run in the presence of time-dependent edge weights. However, would it be not good enough to replace the backward search by a backward BFS? Of course, we can apply stall-on-demand only to the forward search then. Also, we can not stop forward and backward search based on an upper bound B as in Line 19 of Algorithm 5.4. But then it is pointless to perform forward and backward search in an alternating manner. So, we perform forward and backward search separately. The resulting simplified query algorithm always takes at least 3.1 times longer than the TCH-based EA query for *Germany* (see Table 5.6). For *Western Europe high* the TCH-based EA query is only more than 2.2 times faster. Supposedly, this is because of the larger relative delay which negatively affects the backward search during the first phase of this algorithm (i.e., during the bidirectional search). More precisely, the number of overlapping travel time intervals during the backward TTP interval search may be so large that comparatively few predecessors are ruled out (see Lines 9 and 10 of Algorithm 4.3). Also, stall-on-demand is likely to fail more often.

Tables 5.7 and 5.8 take a closer look at the behavior of EA and TTP queries on TCHs, ATCHs, Min-Max-TCHs, and inexact TCHs. For ATCH-based EA queries (see Section 5.4.2) the parameter ε provides a tradeoff between running time and space usage. For ATCH-based TTP queries (see Section 5.4.3) we observe a minimum running time for ε near $1\,\%$. Our interpretation is that there is a tradeoff between the running time of the thinning phase, which includes an approximate TTP search, and of the rest of ATCH-based TTP query; that is, of bidirectional TTP interval search, corridor expansion, and corridor contraction. For greater ε the thinning phase needs less time because the processed approximate TTFs have lower complexity. But the effect of thinning, pruning and stall-on-demand is smaller then. For smaller ε the thinning phase needs more time, but thinning, pruning and stall-on-demand have greater effect. The observed numbers of touched edges and bend points confirm this interpretation.

The running times of EA and TTP query using the Min-Max-TCH representation of *Germany midweek* are smaller than for the ATCH with $\varepsilon = 10\,\%$. At first glance this seems surprising. But note that Min-Max-TCHs use the lower and upper bounds $\min f$ and $\max f$, respectively, of *exact* TTFs f. ATCHs, in contrast, use the lower and upper bounds $\min \overline{f}$ and $\max \underline{f}$. Also, with the relatively large $\varepsilon = 10\,\%$ it is likely that the impact of thinning and stall-on-demand gets worse. Storing the exact bounds $\min f$ and $\max f$ in the ATCH may lower the running times of ATCH-based EA query, especially for larger values of ε. However, we have not tried that. In contrast to *Germany midweek*, Min-Max-TCHs do not work quite as well for *Western Europe high*: The EA query times are significantly worse

		space of TCH			*tchEaQuery*		backward BFS	
road network	TTF set	total [B/n]	overhead [B/n]	overhead factor	avg. time [ms]	speed up	avg. time [ms]	speed up
Germany	Mon	1 001	906	10.5	0.62	1 281	2.05	386
	mid	995	899	10.4	0.62	1 286	1.98	401
	Fri	833	745	9.5	0.57	1 378	1.81	433
	Sat	401	337	6.2	0.44	1 725	1.49	513
	Sun	265	210	4.8	0.41	1 847	1.34	563
Western Europe	high	599	523	7.9	1.47	1 826	3.36	797
	med	187	140	4.0	1.02	2 473	3.60	697

Table 5.5. How EA queries with TCHs performs in time and space. We measure running time of the TCH-based EA query (*tchEaQuery*, see Algorithm 5.4) for *Germany* and *Western Europe* for all our TTF sets. We also measure the running time of a modified EA query, where the backward search (i.e., the TTP interval search during the bidirectional phase of Algorithm 5.4) is replaced by BFS ("backward BFS"). Speedups are calculated with respect to the one-to-one version of time-dependent Dijkstra (see Section 4.2.1). The space usage of the underlying TCH structures is reported in terms of total space usage and also in terms of space overhead (compared to the original road network). The overhead is reported both as an absolute value and as a growth factor. By "B/n" we abbreviate bytes per node.

		space of ATCH			*atchEaQuery*		*atchTtpQuery*	
road network	TTF set	total [B/n]	overhead [B/n]	overhead factor	avg. time [ms]	speed up	avg. time [ms]	speed up
Germany	Mon	206	111	2.2	1.14	695	29.23	?
	mid	208	112	2.2	1.15	689	33.01	?
	Fri	188	100	2.1	1.06	742	24.56	?
	Sat	127	62	2.0	0.69	1 111	5.31	?
	Sun	100	45	1.8	0.61	1 231	3.99	?
Western Europe	high	192	116	2.5	3.16	848	453.02	?
	med	84	37	1.8	2.16	1 164	263.58	?

Table 5.6. How EA and TTP queries with ATCHs ($\varepsilon = 2.5\,\%$) perform in time and space. We report the running times of ATCH-based EA query (*atchEaQuery*, see Algorithm 5.15) and ATCH-based TTP query (*atchTtpQuery*, see Algorithm 5.20) for *Germany* and *Western Europe* with all our TTF sets. Speedups can only be reported for EA queries. For TTP queries speedups are unknown ("?"), because plain TTP search (see Section 4.2.2) is very slow and does not terminate within any reasonable time for *Germany* and *Western Europe*. The rest of the nomenclature is as in Table 5.5.

		space		time		deleteMin		edges		evals		rel. error	
structure, algorithm	ε [%]	total [B/n]	growth factor	avg. [ms]	speed up	avg. #	speed up	avg. #	speed up	avg. #	speed up	max. [%]	avg. [%]
						Germany midweek							
TCH, 5.4	-	995	10.4	0.62	1 288	520	4 612	5 820	950	1 271	162	0.00	0.00
inexact	0.0	995	10.4	0.59	1 339	639	3 755	7 101	779	76	2 686	0.00	0.00
TCH,	0.1	286	3.0	0.56	1 425	642	3 737	7 138	775	78	2 651	0.10	0.02
	1.0	214	2.2	0.57	1 402	654	3 669	7 271	761	85	2 432	1.01	0.27
5.24	2.5	172	1.8	0.59	1 356	668	3 594	7 429	745	92	2 234	2.44	0.79
	10.0	113	1.2	0.83	958	898	2 674	10 109	547	223	920	9.75	3.84
ATCH,	0.1	309	3.2	0.99	807	558	4 299	7 281	760	3 182	65	0.00	0.00
	1.0	239	2.5	1.04	765	588	4 080	7 993	692	3 553	58	0.00	0.00
5.15 +	2.5	208	2.2	1.15	691	625	3 841	9 168	603	4 172	49	0.00	0.00
5.19	10.0	163	1.7	2.11	377	841	2 854	18 947	292	8 870	23	0.00	0.00
	∞	118	1.2	1.24	643	638	3 761	16 212	341	4 043	51	0.00	0.00
						Western Europe high							
TCH, 5.4	-	599	7.9	1.47	1 823	1 021	8 847	13 681	1 563	2 482	276	0.00	0.00
inexact	0.0	599	7.9	2.53	1 061	1 715	5 266	24 274	881	1 485	460	0.00	0.00
TCH,	0.1	239	3.1	2.32	1 155	1 722	5 245	24 389	877	1 498	456	0.15	0.02
	1.0	195	2.6	2.40	1 118	1 782	5 069	25 361	843	1 624	421	1.50	0.20
5.24	2.5	175	2.3	2.57	1 043	1 875	4 817	26 948	794	1 836	372	3.37	0.48
	10.0	144	1.9	2.40	1 115	1 801	5 016	25 692	832	1 681	407	16.21	2.88
ATCH,	0.1	258	3.4	2.20	1 216	1 142	7 907	16 693	1 281	6 420	107	0.00	0.00
	1.0	208	2.7	2.51	1 068	1 223	7 384	20 336	1 052	7 819	87	0.00	0.00
5.15 +	2.5	192	2.5	3.16	848	1 351	6 684	27 583	775	10 500	65	0.00	0.00
5.19	10.0	165	2.2	6.85	391	1 850	4 882	74 315	288	26 911	25	0.00	0.00
	∞	100	1.3	14.35	187	1 690	5 344	207 099	103	51 023	13	0.00	0.00

Table 5.7. Behavior of EA queries with TCHs, ATCHs, and inexact TCHs as underlying hierarchical structure. ATCHs and inexact TCHs are evaluated for different relative errors ε. To denotes Min-Max-TCHs we write $\varepsilon = \infty$. Memory consumption is reported as total space usage in bytes per node ("B/n") and as growth factor compared to the original road network. Speedups and relative ("rel.") errors are reported with respect to the one-to-one version of time-dependent Dijkstra. Maximum and average, respectively, are denoted by "max." and "avg.". Exact EA queries with TCHs are performed using Algorithm 5.4. Exact EA queries with ATCHs are performed using Algorithm 5.15, where shortcuts are expanded on demand as specified in Algorithm 5.19 ("5.15 + 5.19"). Inexact EA queries with inexact TCHs use Algorithm 5.24. An inexact TCH with $\varepsilon = 0$ is an exact TCH but used with Algorithm 5.24, computing exact results in this case.

(though still not bad). But for TTP queries this variant is quite slow and we do not report any results.

For inexact TTP queries (see Section 5.5.2) on *Germany midweek* we observe the smallest running time for ε around 2.5 %. A tradeoff between the quality of the conservative bounds (i.e., $\min\{\min f, \min \tilde{f}\}$ and $\max\{\max f, \max \tilde{f}\}$) and the complexity of the inexact TTFs is a possible explanation: For smaller ε the

		space		time	deleteMin	edges	points	rel. error	
structure, algorithm	ε [%]	total [B/n]	growth factor	avg. [ms]	avg. #	avg. #	avg. #	max. [%]	avg. [%]
				Germany midweek					
TCH, 5.8	-	995	10.4	993.73	570	6 805	16 717 022	0.00	0.00
inexact	0.0	995	10.4	78.30	647	7 179	840 551	0.00	0.00
TCH,	0.1	286	3.0	5.54	650	7 218	50 812	0.10	0.02
	1.0	214	2.2	2.61	662	7 358	20 928	1.03	0.27
5.10	2.5	172	1.8	1.79	677	7 524	12 580	2.44	0.79
	10.0	113	1.2	2.17	924	10 374	12 759	9.69	3.84
ATCH,	0.1	309	3.2	32.83	651	27 563	466 251	0.00	0.00
	1.0	239	2.5	29.99	675	29 950	449 012	0.00	0.00
5.20	2.5	208	2.2	33.00	701	34 008	498 837	0.00	0.00
	10.0	163	1.7	97.27	889	86 127	1 434 780	0.00	0.00
	∞	118	1.2	70.90	579	55 214	1 050 447	0.00	0.00
				Western Europe high					
TCH, 5.8	-	599	7.9	4 077.67	1 132	18 176	54 617 858	0.00	0.00
inexact	0.0	599	7.9	1 956.93	1 866	26 879	17 704 460	0.00	0.00
TCH,	0.1	239	3.1	193.88	1 882	27 060	1 918 590	0.15	0.02
	1.0	195	2.6	102.73	1 953	28 267	978 794	1.37	0.20
5.10	2.5	175	2.3	87.34	2 067	30 276	834 996	3.28	0.48
	10.0	144	1.9	35.93	1 970	28 627	426 327	14.69	2.88
ATCH,	0.1	258	3.4	551.71	1 875	160 496	5 195 293	0.00	0.00
	1.0	208	2.7	377.06	1 960	189 115	3 481 885	0.00	0.00
5.20	2.5	192	2.5	452.82	2 107	256 981	4 178 386	0.00	0.00
	10.0	165	2.2	2 380.66	2 536	1 340 823	24 100 717	0.00	0.00

Table 5.8. Behavior of TTP queries with TCHs, ATCHs, and inexact TCHs as underlying hierarchical structure. The nomenclature is as in Table 5.7 but reporting the number of processed bend "points" and not of TTF evaluations. No speedups are reported, because plain TTP search is so slow that no measurements could be performed in reasonable time. Exact TTP queries with TCHs are performed using Algorithm 5.8, exact TTP queries with ATCHs using Algorithm 5.20, and inexact TTP queries with inexact TCHs using Algorithm 5.10. An inexact TCH with $\varepsilon = 0$ is an exact TCH but Algorithm 5.10 is used computing exact results in this case.

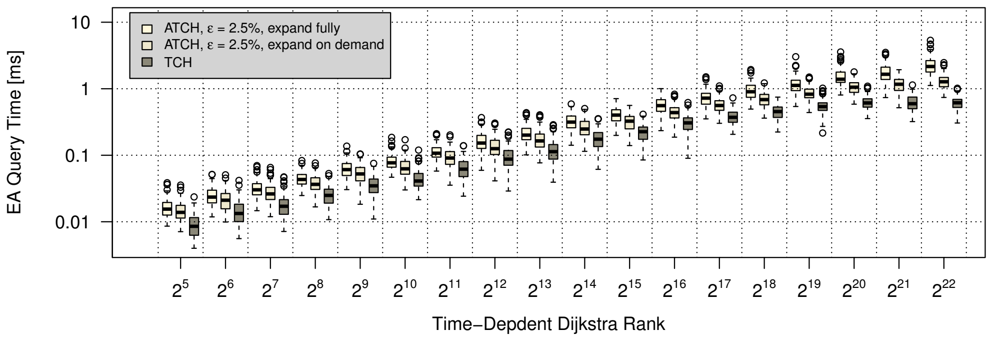
conservative bounds are nearer to the exact bounds (i.e., $\min f$ and $\max f$) than for larger ε. As a consequence, stall-on-demand and pruning have more effect during the bidirectional TTP interval search. But for smaller ε the complexity of the inexact TTFs increases. The number of edges and bend points touched during the computation support this interpretation.

For inexact TTP queries on *Western Europe high* we see a clear correspondence between space and running time. This seems to be natural, as the relative delay is much larger there than for *Germany midweek*, and this makes the impact of pruning based on conservative bounds less effective. As a result, the running time is governed rather by the complexity of the inexact TTFs than by the quality of the conservative bounds. This interpretation is again supported by the number of processed edges and bend points.

Obviously, decreased memory usage directly corresponds to decreased accuracy for inexact TTP queries. So, as space and running time are related, we get a tradeoff between accuracy and running time for this kind of queries. For inexact EA queries (see Algorithm 5.24) on Germany, the running time changes only slightly with the space usage, except for $\varepsilon = 10\,\%$. A possible reason is that inaccurate lower and upper bounds affect the quality of pruning considerably in this case. On *Western Europe high*, an ε of $10\,\%$ is less harmful with respect to running time. The tradeoff between space usage and accuracy remains. Both on *Germany midweek* and *Western Europe high*, the observed maximum errors are small for small ε, and the average errors are even smaller. In theory, however, one can easily construct inputs where errors could get much larger than ε.

In Section 5.4.2 we claim that ATCH-based EA query runs faster if we expand the shortcuts in S^Y only on demand (i.e., Algorithm 5.15 invoking Algorithm 5.19) instead of fully expanding the whole corridor S^Y before we perform time-dependent Dijkstra (i.e., Algorithm 5.15 invoking Algorithm 5.18). Moreover, in Section 5.4.3 we claim that corridor contraction (see Algorithm 5.23) greatly accelerates TTP queries on ATCHs. Figure 5.13 shows that both are really the case. It displays the distribution of running times of EA and TTP queries on *Germany midweek* using a methodology by Sanders and Schultes (see Section 6.4 in their article [75]): For $i = 5..22$ we look at a bulk of 100 queries with the property that the one-to-one version of Dijkstra's algorithm settles the destination node as the 2^i-th node (2^i is called the *Dijkstra rank*). In case of EA queries, "Dijkstra rank" refers to time-dependent Dijkstra search, in case of TTP queries to Dijkstra search with constant travel cost (i.e., the TTF f of an edge $u \to_f v$ is replaced by $\min f$). Note that we repeat this experiment three times; again, to prevent accidental outliers. Then, for every query, we use the median of the three available measured running times as result.

For TTP queries on ATCHs, Figure 5.13 displays two different variants. In the first variant, the corridor $S \subseteq G$ constructed at the beginning of the third phase is not contracted by invoking Algorithm 5.23, but the one-to-one version of plain TTP search (see Section 4.2.2) is performed in S. In the second variant, the corridor $S \subseteq G$ is contracted using Algorithm 5.23 as originally specified in Algorithm 5.20. This demonstrates that contracting the corridor S is really faster. We also display the distribution of running times for the one-to-one version of a plain TTP search (see Section 4.2.2). As this runs very slow, we stop after the average running time exceeds 10 sec. Figure 5.13 also shows the running time distribution for TCH-based EA queries (see Section 5.3.1) and inexact TTP queries using inexact TCHs (see Section 5.5.2).



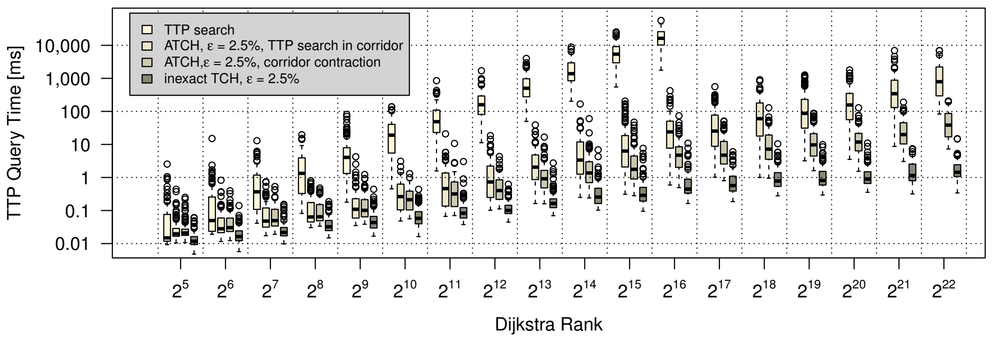


Figure 5.13. Distribution of running times of EA queries (top) and TTP queries (bottom) over Dijkstra rank measured for *Germany midweek*. For every rank a separate distribution is represented as *box plot* each. Bottom and top of each box are the corresponding first quartile Q_1 and the third quartile Q_3, respectively (i.e., each box contains half of the corresponding data points). The stripe inside each box is the corresponding median (i.e., the second quartile Q_2). Each upper/lower whisker is the corresponding maximum/minimum data point not more than $3/2(Q_3 - Q_1)$ away from the median. Every data point outside the whiskers is considered as an outlier and plotted individually.
Top. EA queries based on ATCHs ($\varepsilon = 2.5\,\%$) and TCHs using Algorithm 5.15 and 5.4, respectively. We run Algorithm 5.15 (i.e., for ATCHs) in two different variants. First, the corridor $S^Y \subseteq H$ constructed at the beginning of the third phase is expanded fully and then a time-dependent Dijkstra is performed in the resulting corridor $S \subseteq H$ as specified in Algorithm 5.18 (“expand fully”). Second, the shortcuts contained in $S^Y \subseteq H$ are expanded just the moment they are relaxed as specified in Algorithm 5.19 (“expand on demand”).
Bottom. TTP query based on ATCHs and inexact TCHs with $\varepsilon = 2.5\,\%$ each. ATCHs run with Algorithm 5.20 and inexact TCHs with Algorithm 5.10. The running time of the one-to-one version of plain TTP search (see Section 4.2.2) is also reported. Algorithm 5.20 (i.e., ATCH-based TTP query) runs in two different variants. First, corridor contraction is replaced by a TTP search in the corridor S constructed at the beginning of the third phase (“TTP search in corridor”). Second, the corridor is contracted as specified in Algorithm 5.23 (“corridor contraction”).

Reusing Node Orders. In Section 5.2.3 we claim that a node order, once computed, can be reused to govern the construction of a TCH for the same graph but with a different set of TTFs. Table 5.9 shows the resulting behavior of TCH construction and several query algorithms. It turns out, that this "recycling" works well for *Germany*. For *Western Europe* we have a clear increase of construction

			TCH			ATCH, $\varepsilon = 2.5\,\%$				inexact TCH, $\varepsilon = 2.5\,\%$			
		constr.	space		EA	space		EA	TTP	space		TTP	
TTF set	ordered for	time [h:m:s]	total [B/n]	growth factor	time [ms]	total [B/n]	growth factor	time [ms]	time [ms]	total [B/n]	growth factor	time [ms]	err. [%] max.
Germany													
mid	Mon	0:07:44	1 004	10.5	0.65	208	2.2	1.20	32.90	173	1.8	1.80	2.43
	mid	0:07:33	995	10.4	0.62	208	2.2	1.15	33.01	172	1.8	1.80	2.44
	Fri	0:07:51	1 002	10.5	0.64	209	2.2	1.20	32.86	173	1.8	1.80	2.45
	Sat	0:08:39	1 041	10.9	0.68	211	2.2	1.27	34.33	175	1.8	1.94	2.45
	Sun	0:09:18	1 066	11.2	0.71	213	2.2	1.31	34.86	176	1.8	1.96	2.41
	const	0:10:09	1 147	12.0	0.72	219	2.3	1.32	32.58	180	1.9	1.88	2.45
Sat	Mon	0:03:53	422	6.5	0.45	129	2.0	0.69	5.25				
	mid	0:03:49	422	6.5	0.45	129	2.0	0.69	5.40				
	Fri	0:03:50	418	6.5	0.44	128	2.0	0.69	5.37				
	Sat	0:03:45	401	6.2	0.44	127	2.0	0.69	5.31				
	Sun	0:03:60	416	6.5	0.47	128	2.0	0.72	5.56				
	const	0:04:22	458	7.1	0.48	133	2.1	0.73	5.23				
Sun	Mon	0:03:10	282	5.1	0.40	102	1.9	0.61	3.80				
	mid	0:03:08	283	5.1	0.40	102	1.9	0.60	3.87				
	Fri	0:03:08	279	5.1	0.40	102	1.9	0.60	3.80				
	Sat	0:03:10	273	4.9	0.40	101	1.8	0.61	3.83				
	Sun	0:03:10	265	4.8	0.41	100	1.8	0.61	3.99				
	const	0:03:18	299	5.4	0.41	105	1.9	0.62	3.76				
Western Europe													
high	high	0:51:58	599	7.9	1.47	192	2.5	3.16	453.02	175	2.3	87.27	3.28
	med	1:28:22	723	9.5	1.94	208	2.7	3.93	597.10	189	2.5	120.09	3.19
	const	2:06:06	842	11.1	2.71	214	2.8	5.24	676.69	194	2.5	176.28	2.82
med	high	0:32:35	222	4.7	1.37	88	1.9	2.80	292.13				
	med	0:21:42	187	4.0	1.02	84	1.8	2.16	263.58				
	const	0:55:10	267	5.7	1.69	91	1.9	3.34	271.43				

Table 5.9. Behavior of TCHs, ATCHs, and inexact TCHs when node orders are reused to govern the TCH construction for a different set of TTFs. We report the average running time of TCH-based EA query (see Algorithm 5.4), ATCH-based EA query (see Algorithm 5.15) with shortcut expansion on demand (see Algorithm 5.19), ATCH-based TTP query (see Algorithm 5.20), and inexact TTP queries using inexact TCHs (see Algorithm 5.10). We also report the time needed to construct ("constr.") TCH structures for the different reused orders. For inexact TTP queries we report the maximum relative error ("err."). We also report the space usage of all the TCH, ATCH, and inexact TCH structures. By "const" we denote the node order derived from constant travel costs. The rest of the nomenclature is as in Table 5.7.

time, space usage, and query times, but it still works reasonable. It is, of course, not surprising that this “recycling” of node orders works less well for Western Europe because of its greater average relative delay, which suggests it is more time-dependent.

As an extreme case we perform node ordering only with *constant* travel costs. More precisely, we replace all TTFs of *Germany midweek* and of *Western Europe high* by their minimum and perform node ordering for the resulting graphs. For *Germany* this ordering took 6 min 59 sec and for *Western Europe* 27 min 37 sec. As a result we observe further increased memory usage and query times. Again, this effect seems to be stronger for *Western Europe*. In the past [5] we even used constant cost CHs [43] for node ordering. For a higher percentage of non-constant or stronger varying TTFs, recycling of node orders may not work well enough.

So, depending on the underlying road network and the available sets of TTFs we could save a considerable amount of time by doing node ordering only for the “easy” instances. For the hard instances we would recycle one of the easily obtained orders to govern the TCH construction. Whether all this works well in a specific application context must be found out experimentally by the user.

5.6.4 Comparison with Goal-Directed Techniques

To compare TCHs, ATCHs, and inexact TCHs with some goal-directed time-dependent route planning techniques, look at Table 5.10. These goal directed techniques are *time-dependent core-based routing with ALT* (TD-CALT [24], see page 41) and *time-dependent SHARC* (TD-SHARC [21], see page 41), which can also be combined with ALT (TD-L-SHARC). TD-CALT can only answer EA queries, TD-SHARC can answer EA and TTP queries, though the running times for TTP queries are much larger than in case of the different TCH-based techniques. There are also inexact versions of these goal directed techniques; namely, *approximate TD-CALT*, *heuristic TD-SHARC*, *heuristic TD-L-SHARC*, and *heuristic space efficient TD-SHARC* [14] (see page 42).

For EA queries, we only compare speedups with respect to time-dependent Dijkstra—absolute query times would be unreliable as different machines are used. As plain TTP search takes too long, we are not able to report speedups for TTP queries. Instead, we compare the running times of TTP queries with time-dependent Dijkstra, too. The resulting “speedups”, which we call “relative speed”, enable us to compare the running times of different TTP query algorithms in a machine-independent way. Note that a larger relative speed means smaller running times. As TCH preprocessing works in two stages (node ordering and construction), we always report two preprocessing times in this case, for example, “0:29 / 0:08” (29 minutes for node ordering and 8 minutes construction). Remember that the node ordering already yields a complete TCH structure. So, a separate

		Germany midweek						Western Europe high					
method	ε [%]	prepro-cessing [h:m]	over-head [B/n]	EA speed up	TTP rel. speed	rel. error max. [%]	rel. error avg. [%]	prepro-cessing [h:m]	over-head [B/n]	EA speed up	TTP rel. speed	rel. error max. [%]	rel. error avg. [%]
exact queries													
TCH	-	0:29 / 0:08	899	1 288	10.16	0.00	0.00	3:53 / 0:52	523	1 823	1.37	0.00	0.00
ATCH	0.1	0:29 / 0:08	213	807	24.22	0.00	0.00	3:53 / 0:52	182	1 216	4.86	0.00	0.00
	1.0	0:29 / 0:08	144	765	26.51	0.00	0.00	3:53 / 0:52	132	1 068	7.11	0.00	0.00
	2.5	0:29 / 0:08	112	691	24.10	0.00	0.00	3:53 / 0:52	116	848	5.92	0.00	0.00
	10.0	0:29 / 0:08	68	377	8.18	0.00	0.00	3:53 / 0:52	89	391	1.13	0.00	0.00
	∞	0:29 / 0:08	23	643	11.22	0.00	0.00	3:53 / 0:52	24	187	–	0.00	0.00
TD-CALT	-	0:09	50	280	-	0.00	0.00	1:00	61	47	-	0.00	0.00
TD-SHARC	-	1:16	155	60	0.02	0.00	0.00	6:44	134	70	-	0.00	0.00
TD-L-SHARC	-	1:18	219	238	-	0.00	0.00	6:49	198	150	-	0.00	0.00
inexact queries													
inexact TCH	0.1	0:29 / 0:08	191	1 425	143.48	0.10	0.02	3:53 / 0:52	163	1 155	13.82	0.15	0.02
	1.0	0:29 / 0:08	119	1 402	304.81	1.03	0.27	3:53 / 0:52	119	1 118	26.09	1.50	0.20
	2.5	0:29 / 0:08	77	1 356	445.42	2.44	0.79	3:53 / 0:52	98	1 043	30.68	3.37	0.48
	10.0	0:29 / 0:08	18	958	367.01	9.75	3.84	3:53 / 0:52	68	1 115	74.58	16.21	2.88
approx. TD-CALT	-	0:09	50	804	-	13.84	0.05	1:00	61	624	-	8.69	0.28
heur. TD-SHARC	-	3:26	137	2 164	1.40	0.61	-	22:12	127	1 958	-	1.60	-
heur. TD-L-SHARC	-	3:28	201	3 915	-	0.61	-	22:17	191	2 703	-	1.60	-
sp. eff. TD-SHARC	-	3:48	68	1 177	-	0.61	-	-	-	-	-	-	-
sp. eff. TD-SHARC	-	3:48	14	491	-	0.61	-	-	-	-	-	-	-

Table 5.10. Comparison of different TCH-based and goal-directed methods for exact and inexact EA and TTP queries. Space usage is reported as overhead in byte/node ("B/n"), error as maximum ("max.") relative ("rel.") error. Running times of EA queries have been measured on different machines and are reported only in terms of speedup with respect to time-dependent Dijkstra hence. Running times of TTP queries are reported as relative speed ("rel. speed"), also with respect to time-dependent Dijkstra (greater relative speed means smaller running times). Preprocessing of TCHs has two stages and is reported as in the manner order time / construction time. "L"= combination with ALT, "approx."= approximate, "heur."= heuristic, "sp. eff."= space efficient. Results for goal-directed techniques are taken from the literature [14, 21, 24].

construction phase is not necessary after the node ordering.

Exact Queries. For exact queries, ATCHs dominate TD-SHARC in all respects. TD-CALT is also dominated except for the preprocessing time where TD-CALT is much better. For Western Europe, the advantage of TCH-based techniques over TD-CALT with respect to query time becomes much larger. This is an indication that TCH combined with ALT will not scale well with the road network size.

TCHs versus approximate TD-CALT. For inexact EA queries, approximate TD-CALT has much better speedups than in the exact case. Inexact TCHs, in contrast, have even better speedup but still worse preprocessing time. The memory usage is mostly worse, too. The maximum error of approximate TD-CALT is very large. Regarding that, inexact TCHs are much better, at least if ε is not too large. But, to be fair, the average relative error of approximate TD-CALT is really small. Exact EA queries with Min-Max-TCHs need less memory than the inexact approximate TD-CALT, but the running times are worse. For large ε, inexact TCHs have smaller or, as in case of *Western Europe high*, at least similar memory usage. But the average relative error, and in case of *Western Europe high* also the maximum relative error, is much worse then.

TCHs versus heuristic TD-SHARC. Heuristic TD-SHARC provides faster EA queries than ATCHs with similar memory usage, but with inexact results. Inexact TCHs and heuristic TD-SHARC are both very fast with similar memory usage, although heuristic TD-SHARC has the fastest running times. Space efficient TD-SHARC has the lowest memory usage but at the price of speed. All inexact variants of TD-SHARC have the same relative error for a given road network. The relative error of inexact TCHs, in contrast, depends on ε and can be smaller than in case of TD-SHARC, but the memory usage increases with the accuracy. With respect to the preprocessing time, inexact TCHs are much better than the inexact variants of TD-SHARC.

TTP Queries. Regarding TTP queries, TCH-based techniques are far better than the goal-directed techniques. On *Germany*, ATCHs are up to 1 300 times faster for exact TTP queries, and about 100 times faster for inexact TTP queries if the maximum relative error is 0.1 %. For a maximum relative error of 1.03 %, this rises above 210 times. Note that the *average* relative error of inexact TCHs is considerably smaller than the maximum relative error in both cases. On *Western Europe high*, the goal directed techniques do not provide feasible TTP queries at all. TCH-based techniques, in contrast, provide running times clearly below one second, even in case of exact TTP queries (see Table 5.8).

5.7 References

The whole chapter is actually an extended version of Section 4 to 8 of a journal article published together with Geisberger, Sanders, and Vetter [8]. Many wordings of this article have been used or rephrased, although several things are explained in more detail. Two conference articles [5, 6] and a technical report [7] are indirectly also included, because they form the basis of the journal article.

It is important to note that Christian Vetter made very important contributions to the TCH preprocessing described in Section 5.2, both regarding concepts and implementation. During a student research project [82] he contributed the parallelization of the preprocessing (see Section 5.2.4), which includes the choice of appropriate independent node sets (see Section 5.2.2 on page 190 and Section 5.2.3 on page 198) as well as the definition and maintenance of cost terms that are more suited to time-dependent road networks than the cost terms of the original CHs [44] (see Section 5.2.3 on page 199 to 202).

As a student assistant he contributed the caching of the results of the simulated contractions during node ordering (see Section 5.2.3 on page 202 and 203), the sample search (see Section 5.2.2 on page 195), and the heuristic thinning of the corridor during optimized witness search (see Section 5.2.2 on page 195 and 195). He also adopted the hop limit (see Section 5.2.2 on page 196 and 196) from the original CHs. The parallelization as well as the heuristic thinning of the corridor and the caching of simulation results during optimized witness search are crucial parts of the preprocessing reducing its running time a lot. Without them, the preprocessing would take much more time. Vetter's contributions are part of the implementation we used in the experimental evaluation (see Section 5.6).

The implementation of the Imai-Iri Algorithm [52] has been provided by Sabine Neubauer, who prepared it during her student research project [66]. It is heavily used by our implementation of ATCHs (see Section 5.4.1) and inexact TCHs (see Section 5.5.1). Without Neubauer's implementation, a lot more work would have been necessary.

6

Minimizing Time-Dependent Travel Times with Additional Costs

This chapter adapts the TCHs described in Chapter 5 to work with road networks that not only have time-dependent travel times but also additional time-invariant costs. The resulting generalized TCHs, however, are *heuristic* in the sense that the computed routes are not guaranteed to be optimal. This corresponds to the fact that additional costs make everything much harder. Computing an MC path in this setup is NP-hard and CPs can get more complex than TTPs (see Section 6.1).

The difficulty of computing MC paths also affects TCH preprocessing. So far, we do not know an efficient preprocessing procedure that is able to provide TCH structures with guaranteed existence of optimal up-down-paths. Instead, we describe a generalized TCH preprocessing that is heuristic in the sense that the resulting TCH structures are not exact. Before, we discuss the two main problems that TCH preprocessing has in the presence of additional time-invariant costs. Also, we discuss some preliminary ideas how exact preprocessing might be achieved in the future (see Section 6.2).

Our algorithm for one-to-one MC queries can not guarantee to find optimal routes, as it runs on heuristic TCH structures. It not even guarantees to find the best present up-down-path. Given an exact TCH with guaranteed existence of Pareto prefix-optimal MC up-down-paths, however, the algorithm would be exact. So, exact time-dependent route planning with additional costs would be possible if the preprocessing were able to to provide such a TCH structure (see Section 6.3).

Both, the heuristic preprocessing and the MC queries on the inexact heuristic TCH structures are evaluated experimentally. It turns out that preprocessing can be done in reasonable time and that MC queries can be answered fast with negligible error in practice (see Section 6.4).

6.1 Complexity

The properties of time-dependent road networks with additional time-invariant costs are discussed in Section 3.2. Additional time-invariant costs only seem to be a slight generalization of time-dependent travel times on the first glance. This, however, is not the case, as Section 3.2.1 already suggests. There, we show that the existence of prefix-optimal MC path is not guaranteed in a time-dependent road network G with additional time-invariant costs. More precisely, there may be $s,t \in V$, $\tau_0 \in \mathbb{R}$ such that all (s,t,τ_0)-MC-paths P in G have a prefix $\langle s \to \cdots \to u\rangle \subseteq P$ that is not an (s,u,τ_0)-MC-path. As a consequence, Dijkstra-like single-label search cannot be applied to compute $\mathrm{Cost}(s,t,\tau_0)$ and a corresponding MC path. The reason is that Dijkstra-like single-label searches throw non-optimal intermediate results away. So, an (s,t,τ_0)-MC-path with a prefix $P_u := \langle s \to \cdots \to u\rangle$, where

$$f_{P_u}(\tau_0) + c_{P_u} > \mathrm{Cost}_G(s,u,\tau_0)$$

holds, may get lost. This corresponds to the fact that one-to-one MC queries are NP-hard. This follows from the NP-hardness of a special case of MC queries with much more restricted additional time-invariant costs that has been considered by Ahuja et al. [3]. We report a translation of their proof into the framework used in this thesis (see Section 6.1.1)

One-to-one CP queries, in contrast to one-to-one MC queries, can be answered using a Dijkstra-like single label search; namely, backward CP search (see Section 4.3.5). This algorithm does not seem to be so different from TTP search and some readers may expect similar running times hence. But this is wrong, because a CP can have up to $2^{\Omega(|V|)}$ bend points, which is much more than the $K \cdot |V|^{O(\log |V|)}$ bend points that a TTP can have [36], with K the total number of bend points in G (see Section 6.1.2).

6.1.1 NP-Hardness of Minimum Cost Queries

We consider road networks with time-dependent travel times and additional time-invariant costs; that is, every edge $u \to_{f|c} v$ has a pair $f|c \in \mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$ assigned as edge weight. Ahuja et al. [3] already show the NP-hardness of the very restricted special case that time-dependent edges $u \to_{f|c} v$ have additional time-invariant costs of the form

$$c = \frac{\alpha - \beta}{\beta} \min f$$

with $\alpha > \beta > 0$ chosen globally (i.e., α and β are chosen for the whole graph). This is equivalent to $c = \lambda \min f$, where $\lambda > 0$ is the same for all edges in G (see Section 1.3.2 on page 39 for a short summary).

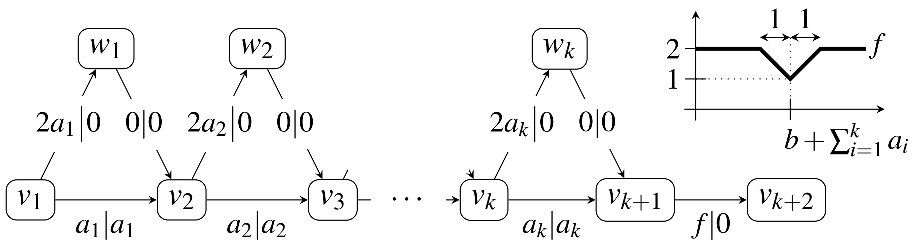


Figure 6.1. A time-dependent graph with additional time-invariant costs encoding an instance $b, a_1, \ldots, a_k$ of the number partitioning problem. All edges have constant TTFs except for $v_{k+1} \to v_{k+2}$ which has the TTF f depicted on the right. That is, f is the only TTF with more than one bend point.

In their proof, Ahuja et al. use a reduction of the NP-hard *number partitioning problem* (see *subset sum problem* in Garey and Johnson [38]). We report a translation of their proof into the context of additional time-invariant costs that can be chosen freely from $\mathbb{R}_{\geq 0}$. The number partitioning problem is defined as follows: Given the numbers $b, a_1, \ldots, a_k \in \mathbb{N}_{>0}$ we ask whether $x_1, \ldots, x_k \in \{0,1\}$ exist with

$$b = x_1 a_1 + x_2 a_2 + \cdots + x_k a_k \ .$$

So, the question is, whether b can be composed of $a_1, \ldots, a_k$, where each a_i can be used at most once.

Theorem 6.1. *Answering one-to-one MC queries is NP-hard, even if there is only a single non-constant TTF with* $\mathrm{O}(1)$ *bend points.*

Proof. This proof is a translation of the proof by Ahuja et al. [3] (see proof of Theorem 2 in their paper with $\alpha = 2$ and $\beta = 1$) into the framework used in this thesis.

Given an instance $b, a_1, \ldots, a_k \in \mathbb{N}_{>0}$ of the number partitioning problem, we construct a time-dependent road network G with additional time-invariant costs as depicted in Figure 6.1. There are exactly 2^k paths from v_1 to v_{k+1}. All these paths have the same total travel cost $2a_1 + \cdots + 2a_k$, but their travel time is not necessarily the same. The TTF of such a path P from v_1 to v_{k+1} is the constant function $f_P \equiv \sum_{i \in X_P} a_i + \sum_{i=1}^{k} a_i$ for some subset $X_P \subseteq \{1, \ldots, k\}$. The subset X_P encodes the path P in the sense that $i \in X_P$ holds if, and only if, P takes the left turn at the node v_i.

In particular, there is a path from v_1 to v_{k+1} of travel time $b + \sum_{i=1}^{k} a_i$ if, and only if, the underlying instance of number partitioning is answered yes. To understand that, consider the following equivalence:

$$\begin{array}{rllrcl} & \text{there is } x_1,\dots,x_k \in \{0,1\} & \text{s.t.} & b & = & x_1a_1+\dots+x_ka_k \\ \Leftrightarrow & \text{there is } X \subseteq \{1,\dots,k\} & \text{s.t.} & b+\sum_{i=1}^k a_i & = & \sum_{i\in X} a_i + \sum_{i=1}^k a_i \\ \Leftrightarrow & \text{there is a path } P \text{ from } v_1 \text{ to } v_{k+1} & \text{s.t.} & b+\sum_{i=1}^k a_i & \equiv & f_P \end{array}$$

Hence, the answer is yes if, and only if, $\text{Cost}_G(v_1, v_{k+2}, 0) = 1 + 2a_1 + \dots + 2a_k$ holds. This is because $f(b+\sum_{i=1}^k a_i) = 1$ is the only minimum of the TTF f. Note that the period of the TTF f has to be chosen sufficiently large.

The construction of the road network from the given instance of the number partitioning problem can be done in $\mathrm{O}(k)$ time. Setting up an adjacency array takes $\mathrm{O}(k)$ time, setting up the sequence

$$\Big\langle \big(b-1+\textstyle\sum_{i=1}^k a_i, 2\big), \big(b+\sum_{i=1}^k a_i, 1\big), \big(b+1+\sum_{i=1}^k a_i, 2\big) \Big\rangle$$

that represents f takes $\mathrm{O}(k)$ time, too. □

The one or another reader may notice that our formulation of the proof is a little simpler than the original; namely, in the sense that all TTFs are constant except for the one of the edge $v_{k+1} \to v_{k+2}$. This is possible, because we can choose the additional time-invariant costs independently from the TTFs. The formulation of Ahuja et al. [3], in contrast, simulates this freedom of choice by inserting artificial minima into the TTFs. Of course, this minima must be so far away from the “time-horizon” of the computation that they do not distort the result of the MC query.

Note that the artificial construction in Figure 6.1 may not be too far away from practice. Routes in road networks may now and then have alternatives with different travel times and different additional time-invariant costs.

6.1.2 Exponential Complexity of Cost Profiles

In the section before we show that one-to-one MC queries in time-dependent road networks with additional time-invariant costs are NP-hard. This implies that one-to-one CP queries are also NP-hard, because MC queries are included in CP queries. About the complexity of CPs, however, a more specific statement is possible; namely, that the worst case complexity of CPs can even be exponential in the size of the road network.

Theorem 6.2. *In a time-dependent road network $G = (V,E)$ with additional time-invariant costs, $\text{Cost}_G(s,t,\cdot)$ can have $2^{\Omega(|V|)}$ bend points, even if there is only a single non-constant TTF with $\mathrm{O}(1)$ bend points in G.*

Proof. Let G be the time-dependent road network with additional time-invariant costs depicted in Figure 6.1 with $a_i := 2^i$ for $1 \leq i \leq k$. Every subset $X \subseteq \{1,\dots,k\}$ corresponds to a path P_X from v_1 to v_{k+1} and vice versa, as explained in the proof of Theorem 6.1. Moreover, all paths P_X have the same constant TCF $C_{P_X} \equiv 2^2 + 2^3 + \cdots + 2^{k+1}$. So, we have

$$\begin{aligned}\mathrm{Cost}_G(v_1, v_{k+2}, \tau) &= \min_{X\subseteq\{1,\dots,k\}} \left\{ f\big(\mathbf{arr}\, f_{P_X}(\tau)\big) + C_{P_X}(\tau) \right\} \\ &= \min_{X\subseteq\{1,\dots,k\}} \left\{ f\big(\tau + \textstyle\sum_{i\in X} 2^i + \sum_{i=1}^{k} 2^i\big) + 2^2 + 2^3 + \cdots + 2^{k+1} \right\} \\ &= \min_{X\subseteq\{1,\dots,k\}} \left\{ f\big(\tau + \textstyle\sum_{i\in X} 2^i + 2^{k+1} - 2\big) + 2^{k+2} - 4 \right\}.\end{aligned}$$

The graph of a TTF $f(\tau + p)$ with $p \in \mathbb{R}$ is simply the graph of f moved to the left by p. So, $\mathrm{Cost}(v_1, v_{k+2}, \cdot)$ is the minimum of 2^k functions that all have 1 as their single global minimum at 2^k different x-values. This means $\mathrm{Cost}(v_1, v_{k+2}, \cdot)$ has at least $2^k = 2^{|V|/2-1} = 2^{\Omega(|V|)}$ bend points. □

Compared to CPs, TTPs are simpler. According to Foschini et al. [36], TTPs have at most $K \cdot |V|^{O(\log |V|)}$ bend points with K being the total number of bend points in G. This suggests that CPs are much harder to obtain than TTPs. It also suggests that backward CP search (see Section 4.3.5) needs more running time than forward and backward TTP search (see Section 4.2.2 and 4.3.1).

Note that the construction in our proof of Theorem 6.2 is very similar to the one Hansen uses to show that bicriteria settings can raise exponentially many Pareto optimal paths [48] (see Section 2.1.3 for a short summary). This already suggests that the two problems are connected. That this is really the case, gets apparent from the idea for an alternative proof of Theorem 6.1 as provided by Sanders [73]. He directly reduces the NP-hard *constrained shortest path problem* (also known as *shortest weight constrained path problem*, see Garey and Johnson [38]) to MC queries with time-invariant additional costs. This problem is the decision variant of computing all Pareto-optimal paths in a graph with two-dimensional edge costs and given start node s and destination node t. W.l.o.g., the reduction considers the first component of the edge costs as constant TTFs and the second component as time-invariant additional costs. Moreover, it adds a new node t' and a new edge $t \rightarrow_{f|0} t'$ with an appropriate TTF f.

6.2 Heuristic Preprocessing

With additional time-invariant costs, we are only able to generate *inexact* heuristic TCH structures so far. The reason is that node contraction with additional costs is more difficult than without. This is because of two major obstacles (see Section 6.2.1). The node contraction procedure that we use with additional costs is heuristic in the sense that travel costs may change in the remaining graph after the removal of the node to be contracted (see Section 6.2.2). The resulting heuristic TCH structures are only guaranteed to contain *some* up-down-path, not necessary an optimal one (see Section 6.2.3). TCH construction and node ordering work very similar as without additional costs, though some minor adaptions are necessary (see Section 6.2.4).

6.2.1 Obstacles of Node Contraction

A TCH structure H generated from a road network G as described in Chapter 5 is guaranteed to contain an (s,t,τ_0)-EA up-down-path for all $s,t \in V, \tau_0 \in \mathbb{R}$. It can thus be used for fast and exact one-to-one EA and TTP queries. This is possible, because time-dependent road networks *without* additional costs fulfill some sort of prefix-optimality; namely, the guaranteed existence of at least one prefix-optimal EA path for every start and destination and all departure times (see Lemma 3.10). This enables the preprocessing to decide whether a shortcut $u \to_{g*f} v$ can be safely omitted only by checking whether the corresponding path $\langle u \to_f x \to_g v\rangle$ is a (u,v,τ)-EA-path for some $\tau \in \mathbb{R}$ in the remaining graph after removal of x (see Section 5.2). Merging a newly inserted shortcut $u \to_{g*f} v$ with an already existing edge $u \to_h v$ is also no problem. We simply replace $u \to_h v$ with $u \to_{\min(h,g*f)} v$.

With additional time-invariant costs, however, it is much more difficult to guarantee the existence of an optimal up-down-path; that is, the existence of an (s,t,τ_0)-MC-path $\langle s \to \cdots \to y \to \cdots \to t\rangle$ for all $s,t \in V, \tau_0 \in \mathbb{R}$, such that $\langle s \to \cdots \to y\rangle$ only goes upward and $\langle y \to \cdots \to t\rangle$ only goes downward in the hierarchy. In the following, we identify two major obstacles. The first one is how to decide efficiently whether a shortcut can be safely omitted during node contraction or not. The second one is that parallel shortcut edges cannot be merged as simply as without additional costs. But before, we define a generalized notion of time-dependent overlay graphs mentioned in Section 5.2.1.

Time-Dependent Overlay Graphs with Additional Costs. Node contraction is the basic operation of TCH preprocessing to obtain the next higher level of the hierarchy each. Without additional time-invariant costs, the contraction of a node x yields a time-dependent overlay graph $G' = (V \setminus \{x\}, E')$ of a graph G; that is,

$$\forall s,t \in V \setminus \{x\}, \tau \in \mathbb{R}: \ \mathrm{EA}_{G'}(s,t,\tau) = \mathrm{EA}_G(s,t,\tau) \tag{6.1}$$

is fulfilled (see Section 5.2.1).[1] With additional time-invariant costs, a generalized time-dependent overlay graph G' would be defined analogously by the condition

$$\forall s,t \in V \setminus \{x\}, \tau \in \mathbb{R} : \ \mathrm{Cost}_{G'}(s,t,\tau) = \mathrm{Cost}_G(s,t,\tau) \ . \tag{6.2}$$

Let $\overline{G} := (\overline{V}, \overline{E}) := (V \setminus \{x\}, \{u \to_{f|c} v \in E \mid u,v \neq x\})$ be the graph that remains after the removal of x and its adjacent edges. To obtain an overlay graph G' we have to determine for which removed path $\langle u \to_{f|c} x \to_{g|d} v\rangle$ a shortcut edge $u \to_{g|d\star f|c} v$ must be added to $\overline{E}$. Adding *all* such shortcuts is likely to produce too dense TCH structures with too slow query times.

First Obstacle: Safely Omitting Shortcuts. With additional costs it is much more difficult than without to decide whether a shortcut can be safely omitted when a node x is contracted. The NP-hardness of one-to-one MC queries (see Theorem 6.1) already suggests this. It also directly transfers to this decision problem.

Lemma 6.3. *Consider the contraction of a node x in a road network with additional time-invariant costs. Deciding whether a shortcut $u \to_{g|d\star f|c} v$ representing a path $\langle u \to_{f|c} x \to_{g|d} v\rangle$ can be safely omitted, is NP-hard.*

Proof. We extend the reduction of the number partitioning problem in the proof of Theorem 6.1 in a straightforward manner. To do so we add a node x as well as two edges $v_1 \to_{0|e} x$ and $x \to_{h|0} v_{k+2}$ to the graph in Figure 6.1 with $e := 2a_1 + \cdots + 2a_k$ and the piecewise linear TTF $h \in \mathscr{F}_\Pi$ defined by the sequence

$$\big\langle (0, 3/2), (1, 5/2), (\Pi - 1, 5/2) \big\rangle \ .$$

Note that $h(0) = 3/2$ is the only minimum of the TTF h. Assuming that the node x is contracted we obtain the following equivalence:

$$\begin{aligned} & \text{shortcut } v_1 \to_{h|e} v_{k+2} \text{ can be safely omitted} \\ \Leftrightarrow\ & \mathrm{Cost}_G(v_1, v_{k+2}, 0) = 1 + e \\ \Leftrightarrow\ & \text{number partitioning answers yes} \ , \end{aligned}$$

which holds because of $C_{\langle v_1 \to_{0|e} x \to_{h|0} v_{k+2}\rangle}(0) = 3/2 + e$ and $\mathrm{Cost}_G(v_1, v_{k+2}, 0) \in \{1+e, 2+e\}$. □

In the following we discuss three possible conditions for safely omitting a shortcut. On the one hand, they are differently strong in the sense of how many

[1]time-dependent overlay graphs generalize the notion of overlay graphs that has been established in the context of constant travel costs [51].

unnecessary shortcuts are not omitted but added to the TCH structure. On the other hand, they are differently difficult to check. All three conditions are *conservative*; that is, necessary shortcut are always added to the TCH. So far it is not clear, however, whether one of these conditions can be used as a basis to provide sufficiently fast exact preprocessing and MC queries.

A Strong, but Hard to Check Condition. From a more practical point of view, the difficulty of safely omitting a shortcut comes from the fact that prefix-optimality is in general not provided in time-dependent road networks with additional costs. When a node x is contracted, this is can result in the situation that $\langle u \rightarrow_{f|c} x \rightarrow_{g|d} v\rangle$ is no (u,v,τ)-MC-path for any $\tau \in \mathbb{R}$ but nevertheless subpath of the only present (s,t,τ_0)-MC-path in G for some $s,t \in V, \tau_0 \in \mathbb{R}$. To decide whether the shortcut $u \rightarrow_{g|d \star f|c} v$ can be safely omitted, it is hence not enough to examine whether the corresponding path $\langle u \rightarrow_{f|c} x \rightarrow_{g|d} v\rangle$ is a (u,v,τ)-MC-path for some $\tau \in \mathbb{R}$. At least, the suffix-optimality of MC path liberates us from considering all (s,t,τ)-MC-paths in G for all $s,t \in V, \tau \in \mathbb{R}$. Instead, it is enough to consider paths starting at u.

Lemma 6.4. *Consider the removal of x and all its adjacent edges from G when a node x is contracted. Then, the following two statements are equivalent:*

1. *For all $s,t \in V \setminus \{x\}$, $\tau \in \mathbb{R}$ we have* $\mathrm{Cost}_{\overline{G}}(s,t,\tau) = \mathrm{Cost}_G(s,t,\tau)$.
2. *For all $u \rightarrow x \in E$, $t \in V \setminus \{x\}$, $\tau \in \mathbb{R}$ we have* $\mathrm{Cost}_{\overline{G}}(u,t,\tau) = \mathrm{Cost}_G(u,t,\tau)$.

Proof. The first statement trivially implies the second one. To show that the second statement implies the first one, assume $\mathrm{Cost}_{\overline{G}}(s,t,\tau) > \mathrm{Cost}_G(s,t,\tau)$ for some nodes $s,t \neq x$ and some $\tau \in \mathbb{R}$. But only MC paths running via node x have changed. So, an (s,t,τ)-MC-path

$$P := \langle s \rightarrow \cdots \rightarrow u \rightarrow x \rightarrow v \rightarrow \cdots \rightarrow t\rangle$$

must be present in G, which implies that its suffix path $R := \langle u \rightarrow x \rightarrow v \rightarrow \cdots \rightarrow t\rangle$ is an $(u,t,\mathbf{arr}\, f_{\langle s \rightarrow \cdots \rightarrow u\rangle}(\tau))$-MC-path in G (see Lemma 3.16). But this means that a path $\overline{R} = \langle u \rightarrow \cdots \rightarrow t\rangle$ with

$$\begin{aligned} C_{\overline{R}}\big(\mathbf{arr}\, f_{\langle s \rightarrow \cdots \rightarrow u\rangle}(\tau)\big) &= \mathrm{Cost}_{\overline{G}}\big(u,t,\mathbf{arr}\, f_{\langle s \rightarrow \cdots \rightarrow u\rangle}(\tau)\big) \\ &= \mathrm{Cost}_G\big(u,t,\mathbf{arr}\, f_{\langle s \rightarrow \cdots \rightarrow u\rangle}(\tau)\big) \\ &= C_R\big(\mathbf{arr}\, f_{\langle s \rightarrow \cdots \rightarrow u\rangle}(\tau)\big) \end{aligned}$$

is present in $\overline{G}$. So, replacing R by $\overline{R}$ in P yields a path $\overline{P} \subseteq \overline{G}$ with

$$\mathrm{Cost}_{\overline{G}}(s,t,\tau) \leq C_{\overline{P}}(\tau) = C_P(\tau) = \mathrm{Cost}_G(s,t,\tau) < \mathrm{Cost}_{\overline{G}}(s,t,\tau)\ ,$$

which is a contradiction. □

Corollary 6.5. *Consider the contraction of a node x. A shortcut edge $u \rightarrow_{g|d * f|c} v$ representing a path $\langle u \rightarrow_{f|c} x \rightarrow_{g|d} v \rangle$ can be safely omitted, if no (u,t,τ)-MC-path of the form $\langle u \rightarrow_{f|c} x \rightarrow_{g|d} v \rightarrow \cdots \rightarrow t \rangle$ exists in G for some $\tau \in \mathbb{R}$.*

So, checking all (u,t,τ)-MC-paths is enough for all $u \to x \in E$, $t \in V$, $\tau \in \mathbb{R}$, which is less difficult than checking all (s,t,τ)-MC-paths in G with arbitrary $s \in V$. Unfortunately, we don't know any better way to check this than answering a one-to-all CP query, which we expect to be very time-consuming.

A Weak Local Condition. To overcome this problem, one could try to establish a condition that is weaker in the sense that less shortcuts can be omitted, but that is easier to check. A relatively obvious weaker and easy to check condition emerges from the observation that a dominated path (see Section 2.1.3) cannot be prefix of an MC path. The two-dimensional costs, where dominance refers to, are pairs of travel time and additional travel costs in this case.

Lemma 6.6. *Consider two paths $P := \langle s \to \cdots \to u \rangle$, $R := \langle s \to \cdots \to u \rangle$ with*

$$f_R(\tau_0) < f_P(\tau_0) \quad \textit{and} \quad c_R < c_P$$

for some $\tau_0 \in \mathbb{R}$; that is, R strictly dominates P. Then, P cannot be prefix path of any (s,t,τ_0)-MC-path.

Proof. Assume an (s,t,τ_0)-MC-path M with prefix P; that is, $M = PU$. Replacing P in M by R yields another path $M' = RU$ from s to t. Utilizing Equation (3.13) we then calculate

$$\begin{aligned}\mathrm{Cost}_G(s,t,\tau_0) = C_M(\tau_0) &= f_U * f_P(\tau_0) + c_U + c_P \\ &> f_U * f_R(\tau_0) + c_U + c_R = C_{M'}(\tau_0) \geq \mathrm{Cost}_G(s,t,\tau_0) \ ,\end{aligned}$$

a contradiction. □

Instantiating the path R with an appropriate MC path for every possible departure time, we obtain a much weaker statement.

Lemma 6.7. *A path $P := \langle u \to \cdots \to v \rangle$ fulfilling both*

- $\mathrm{MCTT}_G(u,v,\tau) < f_P(\tau)$ *and*
- $\mathrm{Cost}_G(u,v,\tau) - \mathrm{MCTT}_G(u,v,\tau) < c_P$

for all $\tau \in \mathbb{R}$ cannot be subpath of any MC path in a road network G.

Proof. For all $\tau \in \mathbb{R}$ consider the certainly existing (u,v,τ)-MC-path R_τ that additionally fulfills $f_{R_\tau}(\tau) = \mathrm{MCTT}_G(u,v,\tau)$ (see Section 3.2.1). We then have $c_{R_\tau} = \mathrm{Cost}_G(u,v,\tau) - \mathrm{MCTT}_G(u,v,\tau)$ and applying Lemma 6.6 we obtain that

P cannot be prefix path of an (u,t,τ)-MC-path for any $\tau \in \mathbb{R}$. But then, it can also not be subpath of any (s,t,τ)-MC-path SPS', because PS' had to be MC path and P a prefix of an MC path hence. □

This directly yields a relatively weak but relatively easy to check condition for safely omitting shortcuts.

Corollary 6.8. *Consider the contraction of a node x. A shortcut edge $u \rightarrow_{g|d \star f|c} v$ representing a path $\langle u \rightarrow_{f|c} x \rightarrow_{g|d} v\rangle$ can be safely omitted, if*

$$\mathrm{MCTT}_G(u,v,\tau) < g * f(\tau) \quad \textit{and} \quad \mathrm{Cost}_G(u,v,\tau) - \mathrm{MCTT}_G(u,v,\tau) < d + c$$

are fulfilled for all $\tau \in \mathbb{R}$.

Both $\mathrm{Cost}_G(u,v,\cdot)$ and $\mathrm{MCTT}_G(u,v,\cdot)$ can be computed by a variant of backward CP search (see Section 4.3.5), which yields $\mathrm{Cost}_G(u,v,\cdot) - \mathrm{MCTT}_G(u,v,\cdot)$. It is possible, however, that this condition is too weak in the sense that too many shortcuts are added during preprocessing. Then, the resulting TCH structures contain too many edges and preprocessing as well as MC queries get too slow. Whether this is the case in practice, must be found out experimentally.

A Stronger Local Condition. To obtain the relatively weak condition in Corollary 6.8 we specialized the statement of Lemma 6.6 by instantiating P with the path $\langle u \rightarrow_{f|c} x \rightarrow_{g|d} v\rangle$ and R with an (u,v,τ)-MC-path R_τ fulfilling $f_{r_\tau}(\tau) = \mathrm{MCTT}_G(u,v,\tau)$ for each $\tau \in \mathbb{R}$. But, in general, there are many more possible paths to instantiate R. These are, in fact, all the Pareto optimal paths from u to v. The resulting condition (see Corollary 6.9) is considerably stronger than Corollary 6.8 but also more difficult to check.

Corollary 6.9. *Consider the contraction of a node x. A shortcut edge $u \rightarrow_{g|d \star f|c} v$ representing a path $\langle u \rightarrow_{f|c} x \rightarrow_{g|d} v\rangle$ can be safely omitted, if a path R_τ exists with*

$$f_{R_\tau}(\tau) < g * f(\tau) \quad \textit{and} \quad c_{R_\tau} < d + c$$

for all $\tau \in \mathbb{R}$.

A method to check this condition could compute a map $W : \mathbb{R} \rightarrow \wp(\mathscr{F}_\Pi \times \mathbb{R}_{\geq 0})$ where $f|c \in W(\tau)$ implies[2]

- that a path $P = \langle u \rightarrow \cdots \rightarrow v\rangle \subseteq G$ exists with $f = f_P$ and $c = c_P$, and
- that no path $R = \langle u \rightarrow \cdots \rightarrow v\rangle \subseteq G$ exists with $f_R(\tau) \leq f_P(\tau)$, $c_R \leq c_P$, and $f_R(\tau) + c_R < f_P(\tau) + c_P$.

[2]With $\wp(\mathscr{F}_\Pi \times \mathbb{R}_{\geq 0})$ we denote the power set of $\mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$.

In other words, the set $W(\tau)$ contains all pairs $f_P|c_P \in \mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$ where P is a non-dominated path from u to v for every departure time $\tau \in \mathbb{R}$. So, we have to answer some kind of one-to-one "Pareto CP query", which seems to be more difficult than computing $\mathrm{Cost}_G(u,v,\cdot)$ and $\mathrm{MCTT}_G(u,v,\cdot)$ as in case of Corollary 6.8.

Second Obstacle: Merging of Parallel Edges. Deciding whether a shortcut can be safely omitted or not is not the only problem that the contraction of a node x has in the presence of additional costs. Merging an already present edge $u \rightarrow_{h|e} v \in \overline{E}$ with a newly inserted shortcut $u \rightarrow_{g|d\star f|c} v$ representing a removed path $\langle u \rightarrow_{f|c} x \rightarrow_{g|d} v \rangle \subseteq G$ is also a problem. More precisely, G' may no longer be an overlay graph of G in the sense of Equation (6.2) if G' simply contains a merged shortcut edge $u \rightarrow_{\min(h|e,\, g|d\star f|c)} v$. This is because the minimum operation on $\mathscr{F}_\Pi^1 \times \mathscr{X}_\Pi$ only preserves information about cheaper paths—although a more expensive path may be part of an MC path whose information is then lost.

So, the one or another MC up-down-path may be missing in the resulting TCH structure. This would be the case even if we had an efficient procedure to safely omit shortcut edges, which we do not so far. Assume, for example, $u \rightarrow_{h|e} v$ is a (u,v,τ)-MC-path in G for all $\tau \in \mathbb{R}$. Also, assume that $\langle u \rightarrow_{f|c} x \rightarrow_{g|d} v \rangle$, which is removed from G when x is contracted, is no MC path itself but subpath of the only (s,t,τ_0)-MC path M in G; that is,

$$M = \langle s \rightarrow \cdots \rightarrow u \rightarrow x \rightarrow v \rightarrow \cdots \rightarrow t \rangle \; .$$

Then, we have $\mathrm{Cost}_{G'}(s,t,\tau_0) > \mathrm{Cost}_G(s,t,\tau_0)$, because the prefix $\langle s \rightarrow \cdots \rightarrow u \rightarrow x \rightarrow v \rangle$ of the unique MC path M is lost; although the shortcut $u \rightarrow_{g|d\star f|c} v$ is not omitted.

To overcome this, one could allow parallel edges, making G' a multigraph[3]. In this case, we would simply add $u \rightarrow_{g|d\star f|c} v$ to E', which already contains $u \rightarrow_{h|e} v$ and maybe other edges from u to v. This, however, is likely to result in quite dense TCH structures, especially because parallel edges may raise even more parallel edges during subsequent node contractions. If, for example, a node x with k incoming parallel edges

$$u \rightarrow_{f_1|c_1} x, \ldots, u \rightarrow_{f_k|c_k} x$$

and ℓ outgoing parallel edges

$$x \rightarrow_{g_1|d_1} v, \ldots, x \rightarrow_{g_\ell|d_\ell} v$$

[3]A *multigraph* $G = (V,E)$ is a directed graph with parallel edges. More precisely, E can contain multiple edge $e_1, \ldots, e_k$ from u to v. This means, E can no longer be a subset of $V \times V$. Instead, edges must be first class citizens. Source and target nodes of edges can be encoded with two maps $src : E \rightarrow V$ and $tgt : E \rightarrow V$ respectively.

is contracted, then $k \cdot \ell$ new parallel shortcut edges of the form $u \rightarrow_{g_i|d_i \star f_j|c_j} v$ may emerge in the worst case. This is likely to slow down preprocessing and MC queries too much. Also, the memory usage of TCH structures may increase considerably.

Storing multiple pairs of TTFs and addtional costs with every merged shortcut edge—that is, something like

$$u \rightarrow_{\langle h_1|e_1,\dots,h_n|e_n\rangle} v$$

with $h_1|e_1,\dots,h_n|e_n \in \mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$—does not really help, because it is only a different representation of the same information. However, several of the pairs $h_i|e_i$ may contribute to an MC path only on a few small departure time intervals, or even never. One could hence store $h_1,\dots,h_n$ as partial real functions. More precisely, if $u \rightarrow_{h_i|e_i} v$ is part of an MC path only if the departure time at u lies in $A_i \subseteq \mathbb{R}$, then we only store the relevant bend points of h_i. Depending on the shortest path structure of the road network this could save a lot of memory. Note that another parallel edge $u \rightarrow_{h_j|e_j} v$ may exist that is part of an MC path for departure times in A_j with $A_i \cap A_j \neq \emptyset$; that is, the domains of the partial functions may overlap.

To determine the intervals where a pair $h_i|e_i \in \mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$ contributes to any MC path, however, we need to find out which non-optimal paths are part of optimal ones. This is actually the same problem as to safely omit shortcut edges. Like in case of safely omitting shortcuts, a sufficiently strong but efficient conservative condition would be helpful.

Observation 6.10. *We do not know a sufficiently strong and feasible conservative method to determine the departure time intervals, where a shortcut edge $u \rightarrow_{g|d \star f|c} v$ contributes to any MC path.*

Here, "conservative" means that the determined intervals contain all departure times where $u \rightarrow_{g|d \star f|c} v$ is part of any MC path. If the determined intervals only contain little more, then the method is "sufficiently strong".

6.2.2 Heuristic Node Contraction

Section 6.2.1 discusses the difficulties of safely omitting a shortcut and of merging parallel edges when a node is contracted. These two important problems that occur during time-dependent preprocessing in the presence of additional time-invariant costs are so far unsolved. As a consequence, we only describe a heuristic version of node contraction in this case. The resulting graph $G' = (V \setminus \{x\}, E)$ after the contraction of a node x does not guarantee Equation (6.2) and is not an overlay graph hence. Instead, G' is only a *heuristic overlay graph*, which means the condition

$$\forall s,t \in V, \tau \in \mathbb{R} : \big(\mathrm{Cost}_{G'}(s,t,\tau) < \infty \iff \mathrm{Cost}_G(s,t,\tau) < \infty\big) \qquad (6.3)$$

is fulfilled. Of course, this condition is very weak compared to Equation (6.2). Our experiments show, however, that the error produced by the resulting heuristic TCH structures stays small in practice (see Section 6.4).

Our heuristic contraction procedure inserts a new shortcut $u \rightarrow_{g|d \star f|c} v$ for a removed path $\langle u \rightarrow_{f|c} x \rightarrow_{g|d} v \rangle$ if

$$\exists \tau \in \mathbb{R} : \mathrm{Cost}_G(u, v, \tau) = (g * f + d *_f c)(\tau) \tag{6.4}$$

is fulfilled. Although this condition is the straightforward generalization of the condition in Equation (5.1) to include additional costs, some necessary shortcuts may get lost. That is, $\mathrm{Cost}_{G'}(s,t,\tau) > \mathrm{Cost}_G(s,t,\tau)$ may hold for the one or another combination $(s,t,\tau) \in V^2 \times \mathbb{R}$ (for details see Section 6.2.1). The shortcuts we insert into $\overline{E}$ to obtain E' are collected in the set

$$\begin{aligned} E_x := \big\{ u \rightarrow_{g|d \star f|c} v \;\big|\; & \langle u \rightarrow_{f|c} x \rightarrow_{g|d} v \rangle \subseteq G \text{ and} \\ & \exists \tau \in \mathbb{R} : (g * f + d *_f c)(\tau) = \mathrm{Cost}_G(u, v, \tau) \big\} , \end{aligned} \tag{6.5}$$

which is analogous to the set E_x as defined in Equation (5.2).

The merging of a newly inserted shortcut $u \rightarrow_{g|d \star f|c} v \in E_x$ and an already present edge $u \rightarrow_{h|e} v \in \overline{E}$ is also heuristic and uses the minimum operation on $\mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$; that is, $u \rightarrow_{h|e} v$ is replaced with the merged edge $u \rightarrow_{\min(h|e, g|d \star f|c)} v$. So, contracting the node x yields the graph $G' = (V \setminus \{x\}, E')$ with

$$\begin{aligned} E' := & \big\{ u \rightarrow_{h|e} v \in \overline{E} \;\big|\; u \rightarrow v \notin E_x \big\} \cup \\ & \big\{ u \rightarrow_{g|d \star f|c} v \in E_x \;\big|\; u \rightarrow v \notin \overline{E} \big\} \cup \\ & \big\{ u \rightarrow_{\min(h|e,\, g|d \star f|c)} v \;\big|\; u \rightarrow_{h|e} v \in \overline{E} \text{ and } u \rightarrow_{g|d \star f|c} v \in E_x \big\} . \end{aligned} \tag{6.6}$$

This is analogous to Equation (5.3). It remains to show that G' is really a heuristic overlay graph.

Lemma 6.11. *Contracting the node x in a time-dependent road network with additional costs preserves the connectivity between the remaining nodes; that is, Equation* (6.3) *holds true.*

Proof. Consider a path $P = \langle s \rightarrow \cdots \rightarrow t \rangle$ in G with $s, t \neq x$. We either have $P \subseteq G'$ or P contains a subpath $\langle u \rightarrow_{f|c} x \rightarrow_{g|d} v \rangle$. If an edge $u \rightarrow v$ exists in G', then we simply replace $\langle u \rightarrow_{f|c} x \rightarrow_{g|d} v \rangle$ with $u \rightarrow v$ in P. Otherwise, we know that $(g * f + d *_f c)(\tau) > \mathrm{Cost}_G(u, v, \tau)$ holds for all $\tau \in \mathbb{R}$. In this case, we choose some $\tau_0 \in \mathbb{R}$ and replace $\langle u \rightarrow_{f|c} x \rightarrow_{g|d} v \rangle$ in P by a surely existing (u, v, τ_0)-MC-path $R_0 \subseteq G$. If R_0 goes via the node x—that is, via a path $\langle u' \rightarrow_{f'|c'} x \rightarrow_{g'|d'} v' \rangle$—we surely know

$$(g' * f' + d' *_{f'} c')\big(\mathbf{arr}\, f_{\langle u \rightarrow \cdots \rightarrow u' \rangle}(\tau_0)\big) = \mathrm{Cost}_G\big(u', v', \mathbf{arr}\, f_{\langle u \rightarrow \cdots \rightarrow u' \rangle}(\tau_0)\big)$$

because of the suffix-optimality of MC paths. But this means that an edge $u' \to v'$ is present in G'. So, replacing $\langle u' \to_{f'|c'} x \to_{g'|d'} v' \rangle$ by the edge $u' \to v'$ in R_0, we end up with a path lying completely in G'.

Consider a path $P' \subseteq G'$. Each edge of P' is either already contained in G or a shortcut $u \to v$ that emerges from a path $\langle u \to x \to v \rangle$. Replacing all such edges $u \to v$ by $\langle u \to x \to v \rangle$ in P yields a path in G. $\square$

Time-Dependent Additional Costs. Merging of shortcuts involves the minimum operation on pairs $h|e$ and $g|d \star f|c$ (see Equation (6.6)). There, it is quite probable that $\min(h|e, g|d \star f|c)$ no longer lies in $\mathscr{F}_\Pi \times \mathbb{R}$ but in $\mathscr{F}_\Pi^1 \times \mathscr{X}_\Pi$, even if $f|c, g|d \star f|c \in \mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$ holds (for details see Section 3.2.2). So, to create a heuristic TCH structure, which is actually a hierarchy of heuristic overlay graphs, we need a node contraction procedure that is able to deal with edge weights taken from $\mathscr{F}_\Pi^1 \times \mathscr{X}_\Pi$ and not only from $\mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$—although all original edges $u \to_{f|c} v \in E$ fulfill $f|c \in \mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$.

For the heuristic node contraction as characterized by Equation (6.6) this is no problem. In fact, we already formulate Equation (6.4), (6.5) and (6.6) as well as the proof of Lemma 6.11 general enough to deal with this situation. Otherwise, we could write them more special. In case of Equation (6.4), for example, it would be

$$\exists \tau \in \mathbb{R} : \mathrm{Cost}_G(u, v, \tau) = g * f(\tau) + d + c \;.$$

It must be noted that we have not proven so far that $\mathrm{Cost}_G(u, v, \tau)$, as used in Equation (6.4) and (6.5) as well as in Lemma 6.11, really exist in the presence of time-dependent additional costs—although we conjecture that this is the case as already said (see Section 3.2.4). Anyway, Lemma 3.23 saves us because the condition formulated in this lemma is most likely fulfilled by time-dependent road networks in practice. So, we require the road network G to fulfill $C_{\min} > 0$ with

$$C_{\min} := \min \left\{ \inf f + \min c \;\middle|\; u \to_{f|c} v \in E \right\} \tag{6.7}$$

in the rest of this chapter. This ensures that an (s, t, τ)-MC-path exists in G for all $s, t \in V, \tau \in \mathbb{R}$ and that $\mathrm{Cost}_G(s, t, \tau)$ as well as $\mathrm{MCTT}_G(s, t, \tau)$ are well defined.

Shortcut Descriptors. Like in case of road network without additional costs, we annotate all edges with shortcut descriptors (see Section 5.2.1 on page 185) to enable the expansion of shortcuts $u \to_{g|d \star f|c} v$ to the paths $\langle u \to_{f|c} x \to_{g|d} v \rangle$ they represent for a given departure time. A formal difference is that the maximal interval where a middle node is valid is no longer closed but left-open in general. To understand that, consider the heuristic merging of two parallel edges $u \to_{f|c} v$ and $u \to_{g|d} v$ with $f|c, g|d \in \mathscr{F}_\Pi^1 \times \mathscr{X}_\Pi$, which means there can be points of discontinuity. Assume $f + c$ has positive discontinuity τ_0 where $g|d$ is continuous with $f+c(\tau_0) < g+d(\tau_0)$ but $f+c(\tau_0) > g+d(\tau_0)$ for all $\tau \in (\tau_0, \tau_0']$ (this can really occur in heuristic TCH structures as explained in Section 6.3.2). Then, a middle node that is valid for $u \rightarrow_{g|d} v$ on $(\tau_0, \tau_0']$ is not necessary valid for the merged edge on τ_0. So, the shortcut descriptors of $u \rightarrow_{f|c} v$ and $u \rightarrow_{g|d} v$ are of the form

$$\big\langle \big((a_0,a_1],x_1\big), \dots, \big((a_{k-1},a_k],x_k\big) \big\rangle$$

and

$$\big\langle \big((b_0,b_1],y_1\big), \dots, \big((b_{\ell-1},b_\ell],y_\ell\big) \big\rangle$$

respectively. Analogous to Equation (5.5), we compute a minimum length sequence

$$\big\langle \big((c_0,c_1],h_1|e_1\big), \big((c_1,c_2],h_2|e_2\big), \dots, \big((c_{n-1},c_n],h_n|e_n\big) \big\rangle \tag{6.8}$$

with $0 = c_0 < c_1 < \cdots < c_n = \Pi$ and $h_1|e_1, \dots, h_n|e_n \in \{f|c,\, g|d\}$, such that

- $h_i|e_i = f|c$ implies $(f+c)(\tau) \leq (g+d)(\tau)$ but $f(\tau) \leq g(\tau)$ if $(f+c)(\tau) = (g+d)(\tau)$, and
- $h_i|e_i = g|d$ implies $(g+d)(\tau) \leq (f+c)(\tau)$ but $g(\tau) \leq f(\tau)$ if $(f+c)(\tau) = (g+d)(\tau)$

for all $\tau \in (c_{i-1}, c_i]$. Again, we refine the partition $(c_0,c_1], (c_1,c_2], \dots, (c_{n-1},c_n]$ by overlaying each interval $(c_{i-1},c_i]$ with one of the two shortcut descriptors depending on $h_i|e_i$. So, for all $i \in \{1,\dots,n\}$, we compute the sequence

$$S_i^{(f|c)} := \big\langle \big((c_{i-1},a_{j_i}],x_{j_i}\big), \big((a_{j_i},a_{j_i+1}],x_{j_i+1}\big), \dots, \big((a_{j_i+n_i-1},c_i],x_{j_i+n_i}\big) \big\rangle$$

with $a_{j_i-1} \leq c_{i-1} < a_{j_i}$ and $a_{j_i+n_i-1} < c_i \leq a_{j_i+n_i}$ if $h_i|e_i = f|c$, and the sequence

$$S_i^{(g|d)} := \big\langle \big((c_{i-1},b_{k_i}],y_{j_i}\big), \big((b_{k_i},b_{k_i+1}],y_{k_i+1}\big), \dots, \big((b_{k_i+m_i-1},c_i],y_{k_i+m_i}\big) \big\rangle$$

with $b_{k_i-1} \leq c_{i-1} < b_{k_i}$ and $b_{k_i+m_i-1} < c_i \leq b_{k_i+m_i}$ if $h_i|e_i = g|d$. The concatenation

$$S := S_1^{(h_1|e_1)} S_2^{(h_2|e_2)} \cdots S_n^{(h_n|e_n)}$$

is then a valid shortcut descriptor for the merged edge $u \rightarrow_{\min(f|c,g|d)} v$. Again, the merged shortcut descriptor S may contain consecutive pairs with the same middle node; that is,

$$S = \big\langle \dots, \big((a,b],x\big), \big((b,c],x\big), \dots \big\rangle$$

is transformed to $S = \langle \dots, ((a,c],x), \dots \rangle$. Again, merged shortcut descriptors are not unique, because $f|c$ and $g|d$ may be equal for some departure intervals.

6.2.3 Heuristic TCH Structures

A heuristic TCH structure is constructed from a time-dependent road network with additional time-invariant costs by performing a sequence of heuristic node contractions in the order given by "$\prec$". Again, "$\prec$" orders the nodes of the road network G by "importance" for routing, with the rough idea that a more important nodes lies on more MC paths. The result is a hierarchy of heuristic overlay graphs

$$G_1 = (V_1, E_1), G_2 = (V_2, E_2), \dots, G_{|V|} = (V_{|V|}, E_{|V|})$$

with $G_1 = G$, $V_{i+1} = V \setminus \{x_i\}$, and $x_1 \prec \cdots \prec x_{|V|}$. Like without additional costs, this hierarchy is never stored explicitly of course. Instead we store the graph $H = (V, E_H)$ with

$$\begin{aligned} E_H := \Big\{ u \rightarrow_{f|c} v \;\Big|\; & \text{there are } k, \ell \text{ with } 0 < k \le \ell < |V| \text{ such that} \\ & \forall i \in \{k, \dots, \ell\} : u \rightarrow_{f_i|c_i} v \in E_i, \text{ but } u \rightarrow v \notin E_{k-1} \cup E_{\ell+1}, \\ & \text{and } f|c = \min(f_k|c_k, \dots, \min(f_{\ell-1}|c_{\ell-1}, f_\ell|c_\ell)\dots) \Big\} \end{aligned} \tag{6.9}$$

and $E_0 = \emptyset$. The graph H together with the information whether an edge $u \rightarrow_{f|c} v$ leads upward (i.e., $u \prec v$) or downward (i.e., $v \prec u$) in the hierarchy is actually what we call a heuristic TCH structure. Again, we define the *upward graph* $H_\uparrow$ and the *downward graph* $H_\downarrow$ by

$$\begin{aligned} H_\uparrow &:= (V, E_\uparrow) := (V, \{u \rightarrow_{f|c} v \in E_H \mid u \prec v\}) \\ H_\downarrow &:= (V, E_\downarrow) := (V, \{u \rightarrow_{f|c} v \in E_H \mid v \prec u\}) \end{aligned} , \tag{6.10}$$

which fulfill $E_\uparrow \cap E_\downarrow = \emptyset$.

Lemma 6.12. *Let H be a heuristic TCH constructed from G and $s, t, \in V$. Then, we have $G \subseteq H$, and t is reachable from s in H if, and only if, this is the case in G.*

Proof. The statement $G \subseteq H$ holds because of $G_1 = G$. Also, heuristic node contraction does not introduce any path between to nodes s, t if t is not reachable from s in G, because of Lemma 6.11. □

Theorem 6.13. *Let H be a heuristic TCH constructed from G and $s, t \in V$. There is an up-down-path from s to t in H if, and only if, t is reachable from s in G.*

Proof. If there is no path from s to t in G, then also not in H; especially, not an up-down-path (see Lemma 6.12). Otherwise, consider a local minimum of a path $P = \langle s = u_1 \rightarrow \cdots \rightarrow u_k = t \rangle$ in G; that is, a node u_i with $u_i \prec u_{i-1}, u_{i+1}$ but $u_i \neq s, t$. Lemma 6.11 tells us that the connectivity was preserved when u_i was contracted during preprocessing. This means H contains a path $\langle u_{i-1} = v_1 \rightarrow v_2 \rightarrow \cdots \rightarrow v_\ell = u_{i+1} \rangle$ whose nodes lie all above u_i. So, repeatedly replacing local minima of P yields an up-down-path in H. □

This proof is a simplified version of the typical correctness proof of CHs as provided by Geisberger et al. [44]. It is so simple, because only connectivity is considered instead of travel costs.

6.2.4 TCH Construction and Node Ordering

TCH construction and node ordering with additional travel costs are similar than without additional travel costs. Algorithm 6.1 shows pseudocode for node ordering. For the case that a node order is already given, we omit the pseudocode. The main differences of node ordering with and without additional costs are

- that edge costs are taken from the set $\mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$ instead of $\mathscr{F}_\Pi$ and
- that E_{new} is computed with respect to Equation (6.5) instead of (5.2).

To obtain a node order and a corresponding heuristic TCH structure, we perform a number of iterations. Every iteration consists of choosing an independent set $I \subseteq V_R$ of nodes in the current remaining graph $R = (V_R, E_R)$ (see Line 9). Afterwards, all nodes $x \in I$ are contracted (see Line 11 to 23). To determine which nodes are contracted next or, in other words, which nodes are put in the independent set I, we proceed as in case of node ordering without additional costs (see Section 5.2.3). More precisely, we assign a non-negative cost value *u.cost* to every node $u \in V$. Whenever a node x is really contracted, we update the cost values *y.costs* of all its neighbors $y \in N^1_R(x)$ by performing a simulated contraction of y (see Line 24). The new cost value is calculated using the same linear combination of cost terms as without additional costs (see Equation (5.17)).

Contracting the nodes $x \in I$ includes the computation of the set E_{new} of shortcuts that we want to insert (see Line 11). To compute E_{new}, we proceed as suggested by Equation (6.5). That means, we iterate over all paths of the form $\langle u \rightarrow_{f|c} x \rightarrow_{g|d} v \rangle \subseteq R$ for all $x \in I$, check whether Equation (6.4) is fulfilled, and add $u \rightarrow_{g|d \star f|c} v$ to E_{new} if we fail to prove the opposite. To check Equation (6.4) we could directly perform the one-to-one version of backward CP search in R to compute $\text{Cost}_R(u, v, \cdot)$. This is, of course, far too slow to be practical. Instead, we utilize optimizations analogous to the ones we use to construct TCHs without additional costs (see Section 5.2.2). Of course, there are some small differences, which we explain in the following.

Optimized Witness Search. The main difference of optimized witness search with and without additional costs is that a backward CP interval search (see Section 4.3.6) and a backward CP search (see Section 4.3.5) are performed instead of a TTP interval search and a TTP search. Note that no forward versions of CP interval search and CP search are available, because MC paths lack prefix-optimality in general (see Section 3.2.1) and Dijkstra-like single-label searches

Algorithm 6.1. Computes a node order and a corresponding heuristic TCH structure for a given time-dependent road network $G = (V, E)$ with additional time-invariant costs.

```
function $orderNodesHeuristicTCH((V,E) : Graph) : (Graph, NodeOrder)$
    $E_H := E$                                        // $G$ is subgraph of $H$
    $(V_R, E_R) := (V, E)$                            // the "remaining graph" $R = (V_R, E_R)$
    $\prec := \langle\rangle$                         // the node order represented as sequence of nodes
    set up random bijective function $noTie : V \to \{1, \dots, |V|\}$
    annotate all edges in $E_H$ with shortcut descriptor $\langle([,\Pi], \perp)\rangle$
    foreach $u \in V$ do simulate contraction of $u$ and initialize $u.cost$
    while $E_R \neq \emptyset$ do
        $I := \{x \in V_R \,|\, \forall u \in N^2_{(V_R,E_R)}(x) \setminus \{x\} : (x.cost < u.cost$ or
                    $(x.cost = u.cost$ and $noTie(x) < noTie(u)))\}$
        append all $x \in I$ to $\prec$
        $E_{\text{new}} := \bigcup_{x \in I} E_x$ with $E_x$ as in Equation (6.5)
        for all $x \in I$ mark all $u \to x \in E_H$ as downward edge in $E_H$
        for all $x \in I$ mark all $x \to v \in E_H$ as upward edge in $E_H$
        for all $x \in I$ remove $x$ and all its incident edges from $(V_R, E_R)$
        foreach $u \to_{g|d\star f|c} v \in E_{new}$ do
            if there is $u \to_{h|e} v \in E_R$ then
                $D :=$ shortcut descriptor of $u \to_{h|e} v$
                $S :=$ merge shortcut descriptors $D$ and $\langle([0,\Pi], x)\rangle$
                replace $u \to_{h|e} v$ by $u \to_{\min(h|e, g|d\star f|c)} v$ in $E_H$ and $E_R$
                annotate $u \to_{\min(h|e, g|d\star f|c)} v$ with merged shortcut descriptor $S$
            else
                add $u \to_{g|d\star f|c} v$ to $E_H$ and $E_R$
                annotate $u \to_{g|d\star f|c} v$ with shortcut descriptor $\langle([0,\Pi], x)\rangle$
        simulate contraction of all $y \in N^1_{V_R,E_R}(I)$ and update $y.cost$ each
    return $((V, E_H), \prec)$
```

do not work hence. However, suffix-optimality of MC-paths is still provided (see Lemma 3.16). So, Dijkstra-like single-label searches can be used if they run in backward direction. More precisely, to show that $\text{Cost}_R(u, v, \tau) < g * f + d *_f c(\tau)$ holds for all $\tau \in \mathbb{R}$, a costly backward CP search is performed after a preceding faster backward CP interval search.

Just like without additional travel costs, witness search can be further accelerated by restricting the backward CP search to a corridor. As corridor we use the transpose predecessor graph of the backward CP interval search; that is, $S := (R^\top(p))^\top \subseteq R$ with p being the predecessor information maintained by the backward CP interval search. The heuristic thinning of the corridor $S \subseteq R$ is also applied analogously. Note that that the resulting thinner corridor $S' \subseteq S$ may not contain the one or another (u, v, τ)-MC-path in R with $\tau \in \mathbb{R}$. So, like without additional costs, some more shortcuts may be added. But this time, this may even improve the quality of the query result. This is the case because additional shortcuts may bring back subpaths of MC paths that would be lost because of the heuristic nature of our node contraction procedure.

Just like without additional costs, we also use a hop limit of 16 when the preceding backward CP interval search runs (see Section 5.2.2 on page 196). Also, we cache the results of all witness searches to avoid multiple simulations of the same node contractions (see Section 5.2.3 on page 202). Sample search (see Section 5.2.2 on page 195), which does not have much impact on the preprocessing without additional costs, is omitted.

More Fine-Grained Parallel Preprocessing. As described in Section 5.2.4, we parallelize witness search in shared memory by assigning nodes we want to contract to threads. This approach also works in the presence of additional travel costs. So, if x is assigned to a thread T, then T computes the set E_x as characterized by Equation (5.2) or (6.5) respectively. In other words, the thread T decides for every path $\langle u \to x \to v \rangle \subseteq R$, whether a shortcut $u \to v$ is inserted or not. If the contraction of the last few nodes takes very long, however, then more and more threads may finish their work while everyone is waiting for a few threads processing the remaining nodes (or even one single thread processing one single node). Such a situation is not unlikely during TCH preprocessing, even without additional costs, because the complexity of TTFs tends to increase a lot as the preprocessing goes on (see Figure 5.11 in Section 5.6.3). In the presence of additional costs it can be worse because the computation of CPs is even more expensive than the computation of TTPs.

To avoid such situations, we make the parallelization more fine-grained if the number of nodes to be contracted gets small; that is, if the number falls below a certain threshold. This means we no longer assign nodes x but paths $\langle u \to x \to v \rangle$ to threads. This can be realized by putting all paths $\langle u \to x \to v \rangle$ into a job queue. The computation of the set E_x can then distributed over more than one thread. Note that we have not evaluated systematically how much this improves the running time of the preprocessing in practice.

6.3 Minimum Cost Queries

This section describes how heuristic TCH structures can be used to answer one-to-one MC queries. The heuristic nature of the TCH structures implies, however, that our query technique only yields inexact results. Note that the algorithm would return exact results if the underlying TCH structure were exact with guaranteed existence of Pareto prefix-optimal MC up-down-paths. Anyway, our experiments show that the error of the query results is negligible in practice.

First, we explain our basic MC query procedure. It is similar to the TCH-based EA query, but with the difference that upward and downward search are time-dependent multi-label searches here (see Section 6.3.1). Then, we proof the correctness of the MC query procedure, including its exactness for the mentioned exact TCH structures—although we are not able to create such TCH structures so far (see Section 6.3.2). Just like like in case of EA queries, MC queries can be answered faster using stall-on-demand, though it is adapted to work correctly with multi-label searches. Note that stall-on-demand may worsen the quality of the computed paths, at least for heuristic TCH structures (see Section 6.3.3). However, the error stays small in practice, as the experimental evaluation in Section 6.4 shows.

6.3.1 Basic MC Querying

Consider a heuristic TCH structure H that has been created from a time-dependent road network G with additional time-invariant costs by the preprocessing procedure described in Section 6.2. Using H, we answer one-to-one MC queries as follows: For a start node s, a destination node t, and a departure time τ_0, we run Algorithm 6.2 on H to obtain an up-down-path with—as we hope—nearly minimal travel cost. Then we invoke Algorithm 5.7 (see Section 5.3.1) to recursively expand the up-down-path to obtain a path in G. Note that Algorithm 5.7 can be applied without any modification, because the expansion of shortcut edges only depends on the departure time and not on the additional travel cost. Together, the computation of the up-down-path and its expansion form a quite fast query procedure.

The computation of the up-down-path (Algorithm 6.2) is the more time-consuming part of this query procedure. Expanding the up-down-path afterwards should only take little time. Algorithm 6.2 generalizes Algorithm 5.4, which computes EA-up-down paths (see Section 5.3.1). Both algorithms have quite similar structure, which means that Algorithm 6.2 also runs in two phases. Again, the first phase is a bidirectional search that only goes upward (see Line 20 to 32), and the second phase goes downward only relaxing edges touched by the backward search of the first phase (see Line 7 to 19). The main differences of Algorithm 6.2 compared to Algorithm 5.4 are

- that the forward search of the bidirectional first phase and the downward search of the second phase are time-dependent multi-label searches (see Al-

Algorithm 6.2. Finds an up-down-path from s to t in a heuristic TCH structure H that has not necessarily minimum cost for departure time τ_0 (though the error is hoped to be small). If a Pareto prefix-optimal (s,t,τ_0)-MC up-down-path is present in H, then such an up-down-path is surely found. As subroutines *tdMultiLabelRelax* (see Algorithm 6.3), *cpIntervalRelax* (see Algorithm 6.4), and *extractPathFromLabelId* (see Algorithm 2.3) are invoked.

```
1  function tchMcQuery($s,t : V$, $\tau_0 : \mathbb{R}$) : Path
2      $L_s[u] := \emptyset$ for all $u \in V$, $L_s[s] := (0, s, \tau_0|0, \bot)$
3      $[q_t[u], r_t[u]] := [\infty, \infty]$ for all $u \in V$, $[q_t[t], r_t[t]] := [0,0]$
4      $p_t[u] := \emptyset$ for all $u \in V$
5      $B := \infty$, $X := \emptyset$                                  // upper bound and candidate set
6      $Q_s := \{((0, s, \tau_0|0, \bot), 0)\}$, $Q_t := \{(t, 0)\}$ : PriorityQueue
7      function downwardSearch() : Path
8          $L_{\text{down}}[u] := \emptyset$ for all $u \in V$
9          $Q_{\text{down}} := \emptyset$ : PriorityQueue                    // PQ for downward search
10         foreach $u \in X$ do
11             foreach $(i, u, \tau|\gamma, i') \in L_s[u]$ do
12                 if $\tau + \gamma + q_t[u] \le B$ then
13                     $Q_{\text{down}}.insert((i, u, \tau|\gamma, i'), \tau + \gamma)$
14                     add $(i, u, \tau|\gamma, i')$ to $L_{\text{down}}[u]$
15         while $Q_{\text{down}} \neq \emptyset$ do
16             $(i, u, \tau|\gamma, i') := Q_{\text{down}}.deleteMin()$
17             if $u = t$ then return extractPathFromLabelId($i$)
18             for $v \in p_t[u]$ do
19                 tdMultiLabelRelax($u \rightarrow_{f|c} v$, $(i, u, \tau|\gamma, i')$, $i_{\text{next}}$, $L_{\text{down}}$, $Q_{\text{down}}$)
20     $\Delta := t$, $i_{\text{next}} := 1$                        // search direction and next unused label id
21     while $(Q_s \neq \emptyset$ or $Q_t \neq \emptyset)$ and $\min\{Q_s.min(), Q_t.min()\} \le B$ do
22         if $Q_{\neg\Delta} \neq \emptyset$ then $\Delta := \neg\Delta$      // change of direction: $\neg s := t$ and $\neg t := s$
23         if $\Delta = s$ then
24             $(i, u, \tau|\gamma, i') := Q_s.deleteMin()$
25         else
26             $u := Q_t.deleteMin()$
27         $M_u := \min\{\sigma + \delta \mid (j, u, \sigma|\delta, j') \in L_s[u]\} \cup \{\infty\}$
28         if $B < \infty$ and $q_t[u] + M_u \le B$ then $X := X \cup \{u\}$
29         $B := \min\{B, r_t[u] + M_u\}$
30         for $u \rightarrow_{f|c} v \in E_\Delta$ do
31             if $\Delta = s$ then tdMultiLabelRelax($u \rightarrow_{f|c} v$, $(i, u, \tau|\gamma, i')$, $i_{\text{next}}$, $L_s$, $Q_s$)
32             else cpIntervalRelax($u \rightarrow_{f|c} v$, $q_t, r_t, p_t, Q_t$)
33     if $B = \infty$ then return $\langle\rangle$ else return downwardSearch()
```

Algorithm 6.3. Edge relaxation procedure as in time-dependent multi-label search (see Algorithm 4.10). Note that the edge weights are taken from the set $\mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$ in the context heuristic TCHs. So, c is a p.w.c. real function. The *Reference* parameters i_{next}, L, Q are necessary to provide context information of the calling time-dependent multi-label search. These are the next unused label id, the label lists associated with the nodes, and the PQ.

```
procedure tdMultiLabelRelax($u \to_{f|c} v$ : Edge, $\ell$ : Label, $i_{\text{next}}, L, Q$ : Reference)
    $(i, u, \tau|\gamma, \cdot) := \ell$
    $\ell_{\text{new}} := (i_{\text{next}}, \mathbf{arr}\, f(\tau)|\gamma + c(\tau), i)$
    if $\ell$ is not dominated by any label in $L[v]$ then
        $Q.insert(\ell_{\text{new}}, \mathbf{arr}\, f(\tau) + \gamma + c)$
        remove all labels dominated by $\ell_{\text{new}}$ from $L[v]$ and $Q$
        add $\ell_{\text{new}}$ to $L[v]$
        $i_{\text{next}} := i_{\text{next}} + 1$
```

gorithm 4.10 in Section 4.4.1) instead of a time-dependent Dijkstra, and

- that the backward search of the bidirectional first phase is a backward CP interval search (see Algorithm 4.8 in Section 4.3.6) instead of a backward TTP interval search.

Remember that Dijkstra-like single label searches can be applied in the presence of additional costs, but in backward direction. This is because suffix-optimality is provided, though prefix optimality is not.

To improve the readability of Algorithm 6.2, the downward search is encapsulated in a nested procedure, just like in case of Algorithm 5.4. To prevent that the pseudocode gets too long, edge relaxations are factored out. The respective pseudocode can be found in Algorithm 6.3 and 6.4.

Phase 1: Bidirectional Upward Search. The forward search is a time-dependent multi-label search running in $H_\uparrow$ and starting from s. The backward search is a backward CP interval search running in $H_\downarrow$ starting from t. So, the backward search is an approximative algorithm (in contrast to the forward search, which is exact). This is necessary, because the arrival time is part of what we want to compute. A backward search that requires a given arrival time cannot be applied hence, just like in case of TCH-based EA queries. Forward and backward search, respectively, only go upward and run in $H_\uparrow$ and $H_\downarrow^\top$ (note that backward searches always run in the transpose graph).

The bidirectional search maintains some data during computation. With respect to the forward search this is the PQ Q_s as well as a label set $L_s[u]$ for every

Algorithm 6.4. Edge relaxation procedure as in backward CP interval search (see Algorithm 4.8). The *Reference* parameters q, r, p, Q are necessary to provide context information of the calling backward CP interval search. These are the node labels, the predecessor information, and the PQ. The procedure is very similar to *ttpIntervalRelax* (see Algorithm 5.6). Only Line 2 is different.

```
1 procedure cpIntervalRelax($u \rightarrow_{f|c} v : Edge, q, r, p, Q : Reference$)
2     $[q_{\text{new}}, r_{\text{new}}] := \big[q[u] + \min(f+c),\, r[u] + \max(f+c)\big]$
3     if $q_{\text{new}} > r[v]$ then return
4     if $r_{\text{new}} < q[v]$ then $p[v] := \emptyset$
5     $p[v] := \{u\} \cup p[v]$
6     if $q_{\text{new}} \geq q[v]$ and $r_{\text{new}} \geq r[v]$ then return
7     $[q[v], r[v]] := \big[\min\{q[v], q_{\text{new}}\},\, \min\{r[v], r_{\text{new}}\}\big]$
8     if $v \notin Q$ then $Q.insert(v, q[v])$
9     else $Q.updateKey(v, q[v])$
```

node $u \in V$. Additional predecessor information is not needed, because a node label $(i, u, \tau|\gamma_u, i') \in L_s[u]$ already contains the id i' of the preceding label on the path from s to u represented by the label. Note that Q_s can contain multiple labels of the same node. Also note that $L_s[u]$ not only contains the unsettled labels (i.e., the labels still in Q_s) but also the settled labels (i.e., the labels already removed from Q_s) of a node u. With respect to the backward search, the bidirectional search maintains the PQ Q_t and for every node $u \in V$ a node label $[q_t[u], r_t[u]]$ as well as a predecessor information $p_t[u]$.

Forward and backward search are, like in case of Algorithm 5.4, performed in an alternating manner. This is controlled by the variable Δ that stores the current search direction (s means forward, t means backward, see Line 22). There is also the cost bound B, which is an upper bound of the travel cost of the cheapest up-down path found so far. It can be updated whenever a nodes label or a node is removed from the PQ, respectively, where the node is reached both by forward and backward search (see Line 29). The candidate node set X stores all nodes where forward and backward search meet. The cost bound B can be used to rule out the one or another candidate node; namely, candidate nodes that are top nodes of up-down-paths that are more expensive than the best up-down path processed so far (see Line 28). The tentative minimum travel cost M_u of a node u is used to rule out candidate nodes and to maintain the cost bound B.

The cost bound B is also used to stop the bidirectional search earlier than Q_s and Q_t running empty. The condition of the while loop interrupts the loop when neither forward nor backward search can contribute to a better up-down-path (see Line 21). After the bidirectional search terminates, we can be sure that the predecessor graphs of forward and backward search together contain one or more up-down-paths from s to t; that is, one or more up-down-paths

$$\langle s \to \cdots \to x \to \cdots \to t \rangle \subseteq H_{\uparrow}(L_s) \cup \left(H_{\downarrow}^{\top}(p_t)\right)^{\top}$$

with x being the top node. Extracting such an up-down-path is what the second phase does.

Phase 2: Downward Search. The second phase, which we call downward search, is a time-dependent multi-label search running on $(H_{\downarrow}^{\top}(p_t))^{\top} \subseteq H_{\downarrow}$; that is, on the transpose predecessor graph of the backward search. The PQ of the downward search is Q_{down} and a label set $L_{\text{down}}[u]$ is assigned to every node $u \in V$. The search starts from the candidate nodes in X, which means the PQ Q_{down} is initialized with the labels in $\bigcup_{u \in X} L_s[u]$. The search stops as soon as the first label of the node t is removed from the Q.down. So, the downward search works like the one-to-one version of multi-label search, although it starts from multiple nodes (namely, from candidate nodes in X). Again, some labels may be ruled out using the cost bound B (Line 10 to 14).

6.3.2 Correctness

Lacking the guaranteed existence of MC up-down-paths in heuristic TCHs, we cannot guarantee that the computed route is an (s, t, τ_0)-MC-path for given start node s, destination node t, and departure time τ_0. We can also not guarantee that Algorithm 6.2 computes the cheapest present up-down-path from s to t for departure time τ_0. The reason is that the TCF $h + e$ of a shortcut edge $u \to_{h|e} v$ in a heuristic TCH is in general only piecewise continuous (i.e., TCFs of shortcuts can have points of discontinuity). As a result, the Pareto prefix-optimality (see Section 4.4.1) of the original road network G is lost in the heuristic TCH. But without Pareto prefix-optimality, forward and downward search are not able to find the cheapest present upward and downward paths respectively. Still, if someone finds a preprocessing procedure creating exact TCH structures that guarantee the existence of Pareto prefix-optimal MC up-down-paths, then we can be sure that the computed up-down-paths represent MC paths in the original road network G. All this is explained in more detail in the rest of this section.

Discontinuity in Heuristic TCH structures. To understand, how heuristic contraction introduces TCFs with discontinuities in spite of the fact that the original road network G only has continuous TTFs and constant additional travel costs, consider the situation depicted in Figure 6.2. A shortcut edge $u \to_{f'|c' * f|c} v$ representing a path $\langle u \to_{f|c} x \to_{f'|c'} v \rangle$ with continuous TTFs and time-invariant additional costs (i.e., $f|c$, $f'|c' \in \mathscr{F}_{\Pi} \times \mathbb{R}_{\geq 0}$) is added during contraction of a node x.

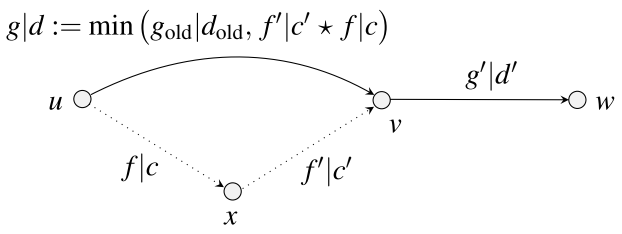

$g|d := \min\left(g_{\text{old}}|d_{\text{old}}, f'|c' \star f|c\right)$
$g'|d'$
u
v
w
$f|c$
$f'|c'$
x

Figure 6.2. How heuristic contraction of nodes introduces piecewise continuous TCFs into a heuristic TCH structure. Contracting the node x, we insert a new shortcut edge $u \rightarrow_{f'|c' \star f|c} v$ that represents the removed path $\langle u \rightarrow_{f|c} x \rightarrow_{f'|c'} v \rangle$ (drawn dotted). The new shortcut is merged with the already present edge $u \rightarrow_{g_{\text{old}}|d_{\text{old}}} v$. Even with $f|c, f'|c', g'|d', g_{\text{old}}|d_{\text{old}} \in \mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$, it is quite likely that the TTF g has points of discontinuity. So, $h+e$ with $h|e := g'|d' \star g|d$ very probably has points of discontinuity, too. This means contracting v after x is likely to introduce a TCF with one or more points of discontinuity.

Obviously, the shortcut has the continuous TTF $f' * f$ and the constant additional cost $c' + c$. Merging the shortcut with an already present edge $u \rightarrow_{g_{\text{old}}|d_{\text{old}}} v$ yields an edge $u \rightarrow_{g|d} v$ with $g|d := \min(g_{\text{old}}|d_{\text{old}}, f'|c' \star f|c)$. In general, the TTF g and the ACF d have points of discontinuity (i.e., $g|d \in \mathscr{F}_\Pi^{\frac{1}{2}} \times \mathscr{X}_\Pi$), though $g+d$ is still guaranteed to be continuous if this is the case for $g_{\text{old}} + d_{\text{old}}$. An example is the situation depicted in Figure 3.7, where the minimum of continuous TCFs results in a continuous TCF, but the corresponding TTF is only piecewise continuous.

Returning to Figure 6.2 assume the node v is contracted and a shortcut is inserted for the path $\langle u \rightarrow_{g|d} v \rightarrow_{g'|d'} w \rangle$. The TCF $h+e$ of the shortcut $u \rightarrow_{h|e} w$ with $h|e := g'|d' \star g|d$ is no longer guaranteed to be continuous then. To understand that, consider

$$\begin{aligned} h+e &= g' * g + d' *_g d = g' \circ \mathbf{arr}\, g + g + d' \circ \mathbf{arr}\, g + d \\ &= (g' + d') \circ \mathbf{arr}\, g + g + d\ . \end{aligned}$$

Although $g' + d'$ and $g + d$ are continuous, this is not the case for the TTF g in general. So, $(g' + d') \circ \mathbf{arr}\, g$ is also not continuous in general and the same holds true for the TCF $h + e = (g' + d') \circ \mathbf{arr}\, g + g + d$. In other words, the linking $h|e := g'|d' \star g|d$ can induce points of discontinuity in $h+e$, because g probably has points of discontinuity. Figure 6.3 shows an example of such a situation where linking results in a TCF with points of discontinuity, although the operands of the link operation represent fully continuous TCFs.

Loss of Pareto Prefix-Optimality. We just explained how piecewise continuous TCFs arise in a heuristic TCH structure H. Now we discuss how these TCFs destroy the Pareto prefix-optimality of H. The actual problem is that discontinuities

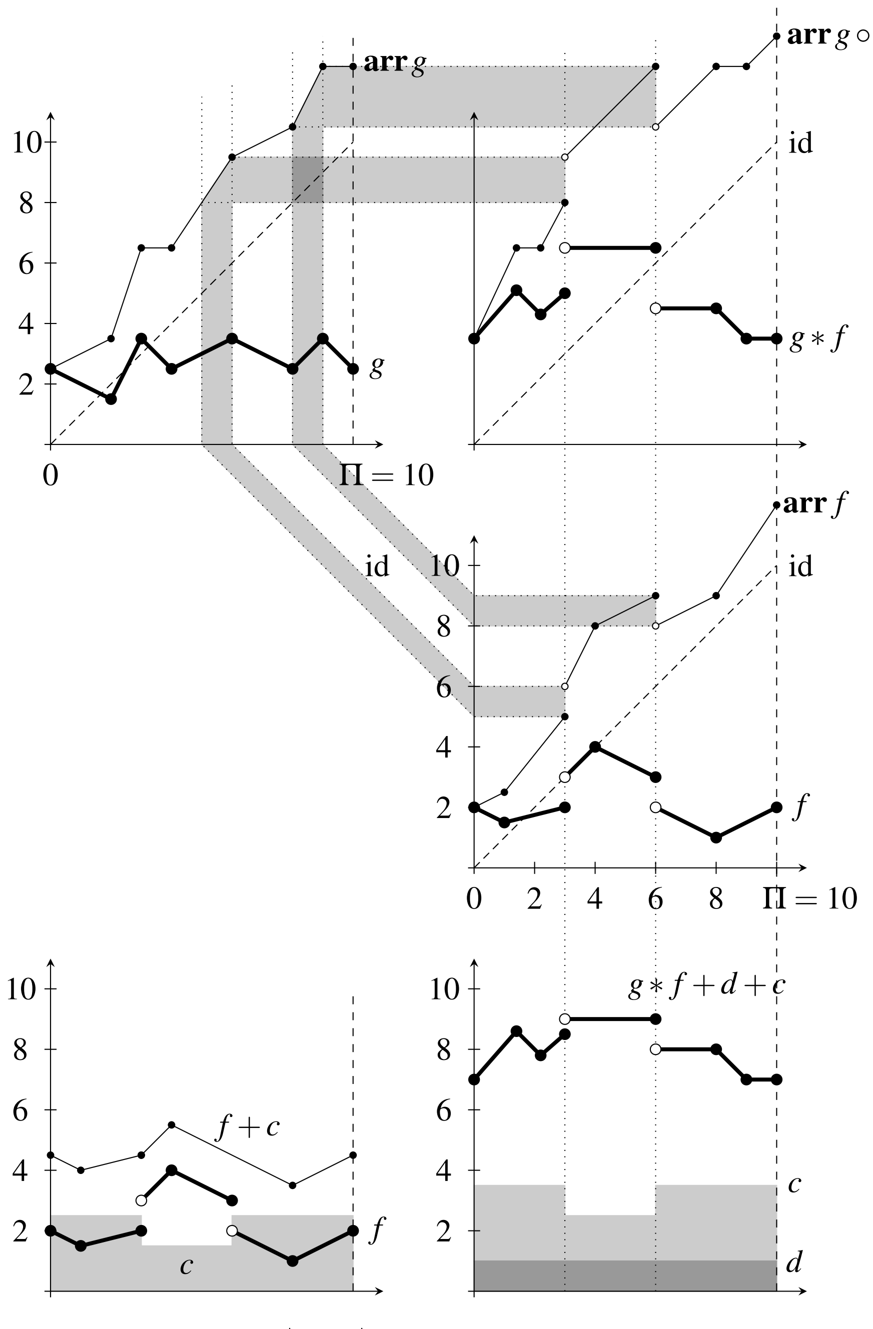


Figure 6.3. How linking $g|d \star f|c$ provokes points of discontinuity in the represented TCF. Consider $f|c \in \mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$ (bottom left), with f,c discontinuous at 3 and 6, but $f+c$ continuous. Also consider $g|d$ with continuous g (top left) and $d \equiv 1$. The TCF $g*f+d*_f c = g*f+d+c$ (bottom right), which is represented by $g|d \star f|c$, is discontinuous at 3 and 6, too. The discontinuity of f at 3 and 6 is inherited by **arr** f and thus by **arr** $g \circ$ **arr** f, $g*f$, as well as by $g*f+c+d$. Note that $g*f$ is visualized in the style of Figure 3.5 with the difference that TTFs have points of discontinuity here.

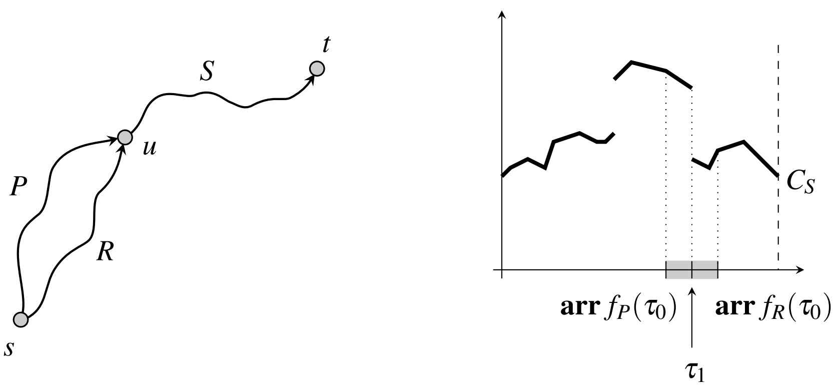


Figure 6.4. How a TCF with a negative discontinuity destroys Pareto prefix optimality. The negative discontinuity of C_S at τ_1 violates the FIFO property and the difference $C_S(\mathbf{arr}\, f_P(\tau_0)) - C_S(\mathbf{arr}\, f_R(\tau_0))$ can get so large, such that $C_S(\mathbf{arr}\, f_R(\tau_0)) + C_R(\tau_0) = C_{RS}(\tau_0) < C_{PS}(\tau_0) = C_S(\mathbf{arr}\, f_P(\tau_0)) + C_P(\tau_0)$ can hold, even if $f_P(\tau_0) < f_R(\tau_0)$ and $c_P(\tau_0) < c_R(\tau_0)$ implying $C_P(\tau_0) < C_R(\tau_0)$ holds.

of TCFs may be negative (see Section 2.1.1). This obviously violates the FIFO property, which is formally fulfilled by all TCFs in the original road network G.

To understand that, consider the situation depicted in Figure 6.4. There are two paths P, R running from s to u as well as a path S running from u to t. Assume $f_P(\tau_0) < f_R(\tau_0)$ and $c_P(\tau_0) < c_R(\tau_0)$ are fulfilled; that is, P strictly dominates R. Further assume $C_S = f_S + c_S$ has a negative discontinuity for departure time $\tau_1 \in [\mathbf{arr}\, f_P(\tau_0), \mathbf{arr}\, f_R(\tau_0)]$. Then, it can happen that

$$C_{PS}(\tau_0) = C_S\big(\mathbf{arr}\, f_P(\tau_0)\big) + C_P(\tau_0) > C_S\big(\mathbf{arr}\, f_R(\tau_0)\big) + C_R(\tau_0) = C_{RS}(\tau_0)$$

holds in spite of the fact that P dominates R for departure time τ_0. This is the case if

$$C_S\big(\mathbf{arr}\, f_P(\tau_0)\big) - C_S\big(\mathbf{arr}\, f_R(\tau_0)\big) > C_R(\tau_0) - C_P(\tau_0)$$

holds; that is, if the jump of the negative discontinuity in C_S is wide enough.

What the Basic MC Query Computes. For the heuristic TCH structures generated by our heuristic preprocessing described in Section 6.2, we can only guarantee that some up-down-path is found by our MC query procedure. This up-down-path is not necessary an (s,t,τ_0)-MC up-down-path.

Lemma 6.14. *Given a TCH structure H generated from G and $s,t \in V$, $\tau_0 \in \mathbb{R}$. Then, Algorithm 6.2 returns an up-down-path from s to t in H.*

Proof. That an up-down-path $\langle s \to \cdots \to x_0 \to \cdots \to t\rangle$ with top node x_0 exists in H is guaranteed by Theorem 6.13. So, x_0 is surely reached by the forward search and $L_s[x_0]$ contains at least one label ℓ_0. Also, x_0 is reached by the backward search. So, t is reachable from x_0 in $(H_\downarrow^\top(p_t))^\top \subseteq H_\downarrow$. Now, consider the "artificial" graph $\hat{H}$ formed by

- the transpose predecessor graph of the backward search $(H_\downarrow^\top(p_t))^\top$,
- the start node s, and
- an "artificial" node y_ℓ and an "artificial" path $\langle s \rightarrow_{\tau_\ell - \tau_0|\gamma_\ell} y_\ell \rightarrow_{0|0} x\rangle$ for every $\ell = (\cdot, x, \tau_\ell|\gamma_\ell, \cdot) \in L_s[x]$ with $x \in X$.

Obviously, t is reachable from s in $\hat{H}$. So, running a time-dependent multi-label search in $\hat{H}$ with start node s and departure time τ_0 surely returns some path from s to t. But such a search is equivalent to the downward search in $(H_\downarrow^\top(p_t))^\top$ starting from the nodes in X. □

As already said, our MC query procedure is optimal in the sense that Pareto prefix-optimal MC up-down-paths are found if present.

Theorem 6.15. *Given a TCH structure H generated from G and $s,t \in V$, $\tau_0 \in \mathbb{R}$. If H contains an (s,t,τ_0)-MC up-down-path $P_0 = \langle s \rightarrow \cdots \rightarrow x_0 \rightarrow \cdots \rightarrow t\rangle$, such that $\langle s \rightarrow \cdots \rightarrow x\rangle \subseteq H_\uparrow$ is Pareto prefix-optimal in $H_\uparrow$ and $\langle x \rightarrow \cdots \rightarrow t\rangle \subseteq H_\downarrow$ is Pareto prefix-optimal in $H_\downarrow$, then Algorithm 6.2 returns such an up-down-path.*

Proof. Let R_0 be the upward part of P_0. Surely, x_0 is reached by the forward search and $L_s[x_0]$ contains a label $\ell_0 = (\cdot, x_0, \mathbf{arr} f_{R_0}(\tau_0)|c_{R_0}(\tau_0), \cdot)$ representing a corresponding Pareto prefix-optimal path from s to x_0 in $H_\uparrow$ (see Corollary 4.68). Also, x_0 is reached by the backward search. There is a Pareto prefix-optimal $(x_0, t, \mathbf{arr} f_{R_0}(\tau_0))$-MC-path S_0 in $(H_\downarrow^\top(p_t))^\top \subseteq H_\downarrow$. To understand that, consider backward CP interval search as a special case of backward approximate CP search (see Algorithm 4.9 in Section 4.3.7) and apply Lemma 4.55, while taking into account that the condition in Line 3 of Algorithm 6.4 is slightly stronger than the analogous condition in Line 14 of Algorithm 4.9. This way we obtain that $(H_\downarrow^\top(p_t))$ contains all (x_0, t, τ)-MC-paths with respect to $H_\downarrow$, which includes Pareto prefix-optimal ones. That also means

$$C_{R_0 S_0}(\tau_0) = C_{R_0}(\tau_0) + C_{S_0}(\mathbf{arr} f_{R_0}(\tau_0)) = C_{P_0}(\tau_0) = \mathrm{Cost}_G(s,t,\tau_0) \ .$$

Again, consider the "artificial" graph defined in the proof of Lemma 6.14. Obviously, $\hat{H}$ contains a path $\hat{S}_0 := \langle s \rightarrow_{f_{R_0}(\tau_0)|c_{R_0}(\tau_0)} y_{\ell_0} \rightarrow_{0|0} x_0\rangle S_0$ with

$$C_{\hat{S}_0}(\tau_0) = C_{S_0}(\mathbf{arr} f_{R_0}(\tau_0)) + C_{R_0}(\tau_0) = \mathrm{Cost}_G(s,t,\tau_0)$$

and there is no cheaper Pareto prefix-optimal path in $\hat{H}$ for departure time τ_0. The one-to-one version of a time-dependent multi-label search in $\hat{H}$ returns the path

$\hat{S}_0$ or another path with travel cost $\mathrm{Cost}_G(s,t,\tau_0)$ hence (see Corollary 4.69). This search is equivalent to the downward search. So, the up-down-path returned by the downward search has travel cost $\mathrm{Cost}_G(s,t,\tau_0)$ for departure time τ_0. □

6.3.3 Stall-on-Demand

The basic query algorithm explained above can be further improved by the pruning technique stall-on-demand, just like without additional costs (see Section 5.3.3). Stall-on-demand has originally been developed in the context of highway node routing [74] and is also used with constant travel cost CHs [44]. Just like in case of EA and TTP queries, stall-on-demand is applied to the first phase of the query algorithm; namely, to the bidirectional phase (see Section 5.3.3). In case of the backward search, which is a backward CP interval search here, stall-on-demand works as in case of backward search of the TCH-based EA query, which is a backward TTP interval search. In case of the forward search, stall-on-demand has to be adapted to work with two-dimensional instead of one-dimensional travel costs.

Stall-on-Demand Forward. The forward search is a time-dependent multi-label search in $H_\uparrow$ starting from s and settles labels instead of nodes. Correspondingly, stall-on-demand does not stall nodes but labels. Whenever a label $\ell_u = (i,u,\tau|\gamma,i')$ of a node u is taken out of the PQ, we iterate over all incoming edges $w \rightarrow_{g|d} u$ of u in $H_\downarrow$ and over all labels $(j,w,\sigma|\delta,j') \in L_s[w]$ to check whether the condition

$$\begin{aligned}\exists\ w \rightarrow_{g|d} u \in E_\downarrow \text{ and } (j,w,\sigma|\delta,j') \in L_s[w] :&\\ \mathbf{arr}\, g(\sigma) < \tau \text{ and } \delta + d(\tau) < \gamma&\end{aligned}$$

is fulfilled. We check, in other words, whether a path of the form

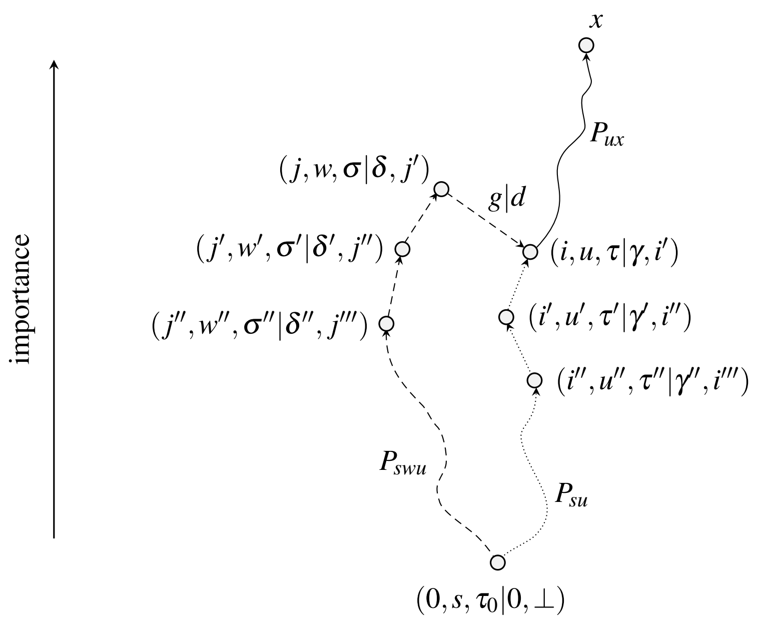
$$P_{swu} := \langle s \rightarrow \cdots \rightarrow w'' \rightarrow w' \rightarrow w \rightarrow_{g|d} u\rangle$$

with $\langle s \rightarrow \cdots \rightarrow w\rangle \subseteq H_\uparrow$ and $w \rightarrow_{g|d} u \in E_\downarrow$ exists that strictly dominates the path

$$P_{su} := \langle s \rightarrow \cdots \rightarrow u'' \rightarrow u' \rightarrow u\rangle \subseteq H_\uparrow$$

represented by the label ℓ_u (see Figure 6.5). If it does, the label ℓ_u is stalled, meaning that no edge $u \rightarrow_{f|c} v$ is relaxed for this label.

Stall-on-demand as just described, rules out the one or another path that is not Pareto prefix-optimal in H, even if it is Pareto prefix-optimal in $H_\uparrow$. This, however, means that stall-on-demand may alter the result of the TCH-based MC query. The reason is that heuristic TCH structures lack Pareto prefix-optimality, as explained in Section 6.3.2. However, there is always a path that is not stalled; that is, some path is always found in the end as the following lemma tells us.



importance
x
P_{ux}
$(j,w,\sigma|\delta,j')$
$g|d$
$(j',w',\sigma'|\delta',j'')$
$(i,u,\tau|\gamma,i')$
$(j'',w'',\sigma''|\delta'',j''')$
$(i',u',\tau'|\gamma',i'')$
$(i'',u'',\tau''|\gamma'',i''')$
P_{swu}
P_{su}
$(0,s,\tau_0|0,\perp)$

Figure 6.5. Stalling of a label $(i,u,\tau|\gamma,i')$ of a node u. When $(i,u,\tau|\gamma,i')$ is taken out of the PQ, we consider all labels $(j,w,\sigma|\delta,j') \in L_s[w]$ with $w \to_{g|d} u \in E_\downarrow$ and stall the label $(i,u,\tau|\gamma,i')$ if $\mathbf{arr}\, g(\sigma) < \tau$ and $\delta + d(\sigma) < \gamma$ is fulfilled. Then, the path $P_{su} \subseteq H_\uparrow$ (drawn dotted) is ruled out, because it is strictly dominated by the path $P_{swu} \subseteq H$ (drawn dashed). But, if the underlying TCH structure lacks Pareto prefix-optimality, then P_{su} may still be prefix path of a path $P_{su}P_{ux}$ that is cheaper than $P_{swu}P_{ux}$; that is, $C_{P_{su}P_{ux}}(\tau_0) < C_{P_{swu}P_{ux}}(\tau_0)$.

Lemma 6.16. *Stalling a label with respect to the forward search, never prevents the TCH-based MC query from finding an up-down-path from s to t.*

Proof. Assume the prefix path $P_{su} := \langle s \to \cdots \to u \rangle \subseteq H_\uparrow$ of an up-down-path

$$\langle s \to \cdots \to u \to \cdots \to x \to \cdots \to t \rangle$$

with top node x is pruned by finding a path $P_{swu}\langle s \to \cdots \to w \to_{g|d} u \rangle$ with $\langle s \to \cdots \to w \rangle \subseteq H_\uparrow$ and $w \to_{g|d} u \in E_\downarrow$. Obviously, x is reachable from w in H, and thus in G. This implies that an up-down-path from w to x is present in H because of Theorem 6.13. Also, w is reachable from s in $H_\uparrow$. So, H contains an up-down-path from s to t that does not contain an edge $u \to v \in E_\uparrow$, and Algorithm 6.2 is still able to find such an up-down-path. □

Note that the above proof would not work if the stalling were propagated to nodes higher up in the TCH structure (as we do it in case of EA queries, see Section 5.3.3). Again, this is because of the lack of Pareto prefix-optimality that we have in heuristic TCH structures. As a consequence, the propagation of stalling can prevent that an up-down-path from s to t is found. To repair this it is not enough that an MC up-down-path exists whose upward part is Pareto prefix-optimal with respect to $H_\uparrow$ and whose downward part with respect to $H_\downarrow$. Instead, we need an MC up-down-path that is Pareto prefix-optimal with respect to H, which is a stronger condition.

Stall-on-Demand Backward. We already said that stall-on-demand with respect to the backward search, which is a backward CP interval search, works in the same way as in case of the backward search of the TCH-based EA query. This means, a node u is stalled if an edge $u \to_{g|d} w \in E_\uparrow$ exists with $q_t[u] > r_t[w] + \max(g+d)$. That stalling a node with respect to the backward search does not prevent us from finding *some* up-down-path from s to t, follows with an analogous argument as in case of the forward search. Note that we do not propagate the stalling to further nodes, just like in case of the forward search.

6.4 Experimental Evaluation

This section reports an experimental evaluation of inexact MC querying with heuristic TCHs as described in this chapter. The underlying test instances (see Section 6.4.1) all relay on a road network instance of Germany with TTFs reflecting midweek high traffic and utilize different sets of additional time-invariant travel costs. After explaining the basic experimental setup (see Section 6.4.2), we present the observed results for preprocessing and querying. The main outcome is that inexact MC queries are fast enough with negligible error; at least, for the considered test instances (see Section 6.4.3).

6.4.1 Input Instances

We use six input instances that consist of the same time-dependent road network instance of Germany, each combined with one of six different sets of time-invariant additional costs.

Time-Dependent Road Network. The underlying time-dependent road network instance of Germany is provided by PTV AG [71] for scientific use. It has about 4.7 million nodes, 10.8 million edges, and TTFs reflecting midweek (i.e., Tuesday till Thursday) traffic collected from historical data; that is, a high traffic scenario with about 7.2 % non-constant TTFs. Table 6.1 summarizes this information and reports the average relative delay including and excluding constant TTFs (for a definition of average relative delay see Section 5.6.1 on page 248). For each edge not only a TTF is available but also the *driving distance*. We use the driving distance as a basis to form the additional constant travel costs as explained below.

Germany midweek								
				avg. relative delay		complexity		
$\|V\|$ [mio]	$\|E\|$ [mio]	space [B/n]	non-const. TTFs [%]	including const. [%]	excluding const. [%]	avg. incl. const. [#]	avg. excl. const. [#]	max. [#]
4.7	10.8	131	7.2	2.6	36.1	2.1	16.3	51

Table 6.1. Some properties of our input road network: number of nodes ("$|V|$") and edges ("$|E|$"), space usage in byte per node ("B/n"), percentage of time-dependent edge weights (i.e., "non-const. TTFs"), as well as the average ("avg.") relative delay and complexity of the TTFs, both including ("incl.") and excluding ("excl.") the constant ("const.") ones. The maximum ("max.") occuring complexity of a TTF is also reported.

This road network instance is in principle the same as the instance *Germany midweek* used in the experimental evaluation of TCHs without additional costs (see Section 5.6). Both versions of *Germany midweek* are extracted from the same raw data provided by PTV AG [71]. The version used in Section 5.6 has been completely provided by Delling. The version used here has been extracted utilizing program code provided by Delling. Any differences should be small.

Table 6.1 reports the total space usage of the input road network when used as underlying graph for time-dependent multi-label A* search. The reader may already have noticed that the space usage is considerably greater than the space usage of *Germany midweek* reported in Table 5.1. This is mainly because the graph data structure used here allows forward and backward search. More precisely, each edge is represented by two data objects instead of a single one; one to enable relaxation in forward direction and one in backward direction. Storing the driving distance of each edge also increases the space usage a little, but much less than storing each edge twice.

Time-Invariant Additional Costs. We use a very simple approximation of energy costs to provide input instances for our experimental evaluation of heuristic TCHs. The idea is that the additional time-invariant travel cost of an edge estimates the energy cost raised by traveling along that edge. With typical gasoline prices we assume that driving 1 km costs 0.1 €. To fix a proportion between energy costs and time spent driving, we must fix a prize for travel time, too. We use different rates of 5 €, 10 €, and 20 € per hour. Using 0.1 sec as unit of time and 1 m as unit of distance we obtain time-dependent total costs of

$$\mathit{time} + \lambda \cdot \mathit{distance} \tag{6.11}$$

where λ has the values 0.72, 0.36, and 0.18 respectively.

To understand how an hourly rate of 5 € corresponds to $\lambda = 0.72$, for example, consider how the cost for traveling a distance of $d \cdot 1\,\mathrm{m}$ in time $\tau \cdot 0.1\,\mathrm{sec}$ is

Germany midweek, energy cost = 0.1 €/km			
motorway tolls	hourly rate	λ_{default}	$\lambda_{\text{motorway}}$
none	5 €/h	0.72	0.72
	10 €/h	0.36	0.32
	20 €/h	0.18	0.18
0.1 €/km	5 €/h	0.72	1.44
	10 €/h	0.36	0.72
	20 €/h	0.18	0.36

Table 6.2. Six input instances derived from *Germany midweek* that all have energy cost of 0.1 €/km, but different hourly rates of 5 €/h, 10 €/h, or 20 €/h, each with or without motorway tolls (i.e., an extra price of 0.1 €/km on motorways). The two-dimensional travel cost of an edge $u \to_f v$ with driving distance d is $f|\lambda d \in \mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$; with $\lambda = \lambda_{\text{motorway}}$ if the edge belongs to a motorway and $\lambda = \lambda_{\text{default}}$ otherwise.

calculated. The travel cost amounts to

$$cost = \frac{5€}{\mathrm{h}} \cdot 0.1\,\mathrm{sec} \cdot \tau + \frac{0.1€}{\mathrm{km}} \cdot 1\,\mathrm{m} \cdot d = \frac{5€}{36000} \cdot \tau + \frac{1€}{10000} \cdot d$$

in this case. It is important to note that MC paths do not change if all travel costs in a graph are scaled by the same factor. It is hence equivalent to deal with any travel cost proportional to *cost*. We choose

$$cost \sim \tau + \frac{36000€}{5€ \cdot 10000} \cdot d = \tau + 0.72 \cdot d\ ,$$

which has the great advantage that travel time and additional time-invariant cost are separated cleanly, just like in Equation (6.11).

Accordingly, the additional constant travel costs are simply the driving distance scaled by the respective value of λ. This yields three input instances of *Germany midweek*, one for each $\lambda \in \{0.72, 0.36, 0.18\}$. An edge $u \to_{f_{uv}} v \in E$ with driving distance d_{uv} gets the two-dimensional edge weight $f_{uv}|\lambda d_{uv} \in \mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$; that is,

$$u \to_{f_{uv}|\lambda d_{uv}} v \in E\ .$$

We also consider the effect of motorway tolls. To do so, we fix an extra price of 0.1 €/km on motorways. This means that edges belonging to motorways have double additional constant edge costs of 0.2 €/km. This can be expressed by using two different values of λ; namely, $\lambda_{\text{motorway}} := 2\lambda$ for edges belonging to a motorway and $\lambda_{\text{default}} := \lambda$ for all other edges. This yields three further instances of *Germany midweek*. Altogether, we have six instances of *Germany midweek*: two instances for every hourly rate of 5 €, 10 €, or 20 €, with and without motorway tolls each. Table 6.2 summarizes all six instances.

6.4.2 Setup

The experimental evaluation is done on a machine with two Core i7 Quad-Cores clocked at 2.67 Ghz with 48 GiB of RAM with Ubuntu 12.04.5 using GCC 4.6.4 with optimization level 3. It is the same machine as used to evaluate TCHs with travel times as travel costs (see Section 5.6.2) but with different versions of Linux and GCC. Running times are always measured using one single thread except of the preprocessing where 8 threads are used. Note that we use the system timer to measure running times. This is different to the experiments described in Section 5.6.3, where we count CPU cycles to measure query times. To avoid unstable query times caused by NUMA effects, we force the CPU core to solely use its local memory. In case of the preprocessing, which runs in parallel, this is not possible. So, the preprocessing times may vary a little.

The performance of the algorithms is evaluated in terms of running time, memory usage, and how often *deleteMin* is invoked. Note that TCH-based MC queries (see Section 6.3) are inexact because of the heuristic nature of the underlying TCH structures (see Section 6.2). For this reason, maximum and average relative error are also reported. To compute the exact results, we use time-dependent multi-label A* search (see Section 4.4.2). To measure the average running time of MC queries we use 10 000 randomly selected start and destination pairs, together with a departure time randomly selected from $[0\text{h}, 24\text{h})$ each. Note that we measure the running time of the 10 000 MC queries only once. This is different from the experiments in Section 5.6.3 where we measure the running times of 1 000 queries three times reporting the average of the median running times as result.

To measure maximum and average relative error as well as the average number of invocations of *deleteMin*, we also run MC queries with randomly selected start, destination, and departure time, but 100 000 instead of 10 000. The memory usage is given in terms of the average *total* space usage of a node (not the overhead) in byte per node. For TCH-based techniques, all figures refer to the scenario that not only the time-dependent travel costs but also the routes have to be determined.

Besides the bunches of 10 000 and 100 000 *random queries* just described, we also perform *rank queries*; that is, for each $i \in \{6, 8, \ldots, 22\}$ we look at a bulk of 1 000 queries with the property that the one-to-one version of time-dependent Dijkstra settles the destination node as the 2^i-th node (2^i is called the *time-dependent Dijkstra rank*). Note that we already use this methodology by Sanders and Schultes (see Section 6.4 in their article [75]) to plot the distribution of EA query times over (time-dependent) Dijkstra rank (see Section 5.6.3 on page 260). Here, we use it to plot the distribution of relative errors over time-dependent Dijkstra rank.

6.4.3 Results

First, we report the results obtained by performing random queries. There, we are not able to report relative errors for the input distances with motorway tolls, as multi-label A* search does not terminate in reasonable time there. For this reason, we perform rank queries to report relative errors for Dijkstra ranks up to 2^{20}. There, multi-label A* search is feasible even with motorway tolls.

Random Queries. Table 6.3 shows the behavior of inexact TCH-based MC queries (see Algorithm 6.2) running on heuristic TCH structures that are created by the preprocessing described in Section 6.2. The underlying inputs are the six input instances derived from *Germany midweek* summarized in Table 6.2. The behavior of two other algorithmic techniques is also shown. These are

- time-dependent multi-label A* search (see Section 4.4.2), which we use as a reference method to obtain exact minimum travel costs, as well as
- inexact TCH-based MC query (see Algorithm 6.2) running on a *travel time TCH*; that is, on a TCH structure created by the preprocessing described in Section 5.2.

Note that the travel time TCHs could be used for exact EA querying with Algorithm 5.4. Also note that the travel time TCHs are not created with the implementation used in Section 5.6. Instead, we use the implementation of heuristic TCH preprocessing with some adaptions. Running the MC query procedure on travel time TCHs is interesting, because we want to know, whether heuristic TCHs actually bring a benefit. Our results show that this definitely the case, as discussed below.

Considering Table 6.3, one sees that motorway tolls make the answering of MC queries much harder, because the observed space usages and running times are much larger than without motorway tolls. Without motorway tolls we get the hardest instance at an hourly rate of 5 €, which corresponds to $\lambda = 0.72$. For higher hourly rates (i.e., for smaller values of λ), things seem to be easier. This is not surprising because smaller values of λ mean that travel time and travel cost are more correlated. With motorway tolls, in contrast, we get the hardest instance at an hourly rate of 10 € (i.e., $\lambda_{\text{default}} = 0.36$ and $\lambda_{\text{motorway}} = 0.72$). A possible explanation is that motorway tolls are negatively correlated to travel time.

Without motorway tolls, multi-label A* has running times similar to Dijkstra's algorithm, as its running time is mainly governed by the running time of the Dijkstra-like backward CP interval search in this case. This follows from the relatively few invocations of *deleteMin* (the invocations raised by the backward CP interval search are not included in Table 6.3). Note that this low number of invocations implies that a more efficient choice of the heuristic function π_t would

Germany midweek												
		space	order	query		rel. error		percentage of rel. error ≥			
method or structure	hourly rate	total [B/n]	time [h:m]	delMin [#]	time [ms]	max. [%]	avg. [%]	1% [%]	0.1% [%]	0.01% [%]	0.001% [%]
energy cost = 0.1 €/km, no motorway toll											
multi-label A*		131	–	271 536	2 155.26	–	–	–	–	–	–
heuristic TCH	5 €/h	1 478	0:35	2 127	3.92	0.11	0.00	0.00	0.00	0.28	1.02
travel time TCH		1 063	0:27	1 207	1.89	14.82	0.68	22.15	63.03	76.76	80.82
multi-label A*		131	–	188 036	2 090.19	–	–	–	–	–	–
heuristic TCH	10 €/h	1 315	0:29	1 774	3.04	0.08	0.00	0.00	0.00	0.05	0.27
travel time TCH		1 063	0:27	1 198	1.89	9.41	0.26	7.69	33.94	49.17	55.53
multi-label A*		131	–	155 489	2 086.46	–	–	–	–	–	–
heuristic TCH	20 €/h	1 209	0:29	1 480	2.48	0.27	0.00	0.00	0.01	0.04	0.13
travel time TCH		1 063	0:26	1 182	1.88	7.38	0.09	2.19	15.31	26.56	32.31
energy cost = 0.1 €/km, motorway toll = 0.1 €/km											
heuristic TCH	5 €/h	1 859	1:10	4 853	12.92						
travel time TCH		1 063	0:25	2 681	3.76						
heuristic TCH	10 €/h	2 033	1:23	10 483	34.03						
travel time TCH		1 063	0:25	2 692	3.66						
heuristic TCH	20 €/h	1 648	0:53	7 161	22.10						
travel time TCH		1 063	0:25	2 534	3.59						

Table 6.3. Behavior of time-dependent MC queries for different hourly rates 5 €/h, 10 €/h, and 20 €/h. With energy costs of 0.1 €/km this corresponds to the λ values 0.72, 0.36, and 0.18 respectively. Results are reported both with additional motorway tolls of 0.1 €/km and without tolls. The reported node order time always refer to parallel ordering with 8 threads in shared memory. Query performance is reported in terms of average running time and average number of removals from the PQ ("delMin", without PQ removals of interval search in case of multi-label A*). Relative ("rel.") error of the query result is reported maximum ("max.") and average ("avg.") relative error. Also, the percentage of routes with a relative error greater or equal 1%, 0.1%, 0.01%, and 0.001% respectively ("percentage of rel. error ≥"). With motorway tolls we cannot report any relative errors or error ratios, because multi-label A* runs too slow to provide exact results in reasonable time in this case.

turn time-dependent multi-label A* search into a pretty fast algorithm (at least without motorway tolls). Hub-labeling [1] may be an appropriate candidate (for a short summary see Section 1.3.1 on page 32). However, with motorway tolls multi-label A* is no longer efficient. Accordingly, Table 6.3 omits the respective errors and error rates as we do not have the exact results in this case.

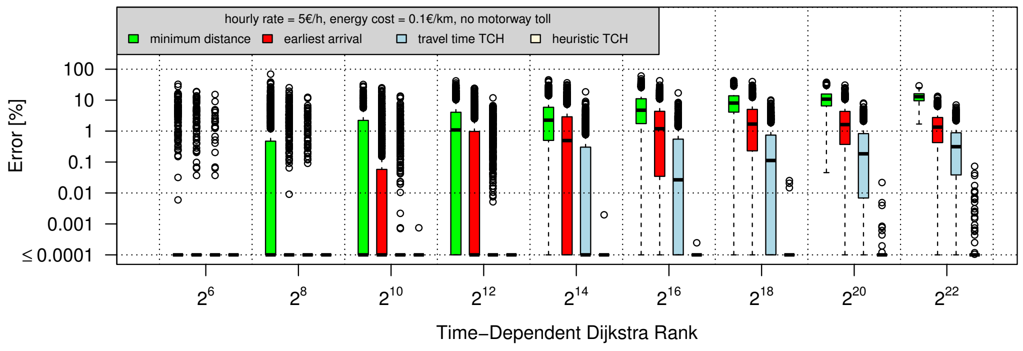
With less than 4 msec and 35 msec, inexact MC queries on heuristic TCHs have much faster query time than multi-label A* search. Though being inexact in theory, the method is practically exact for the kind of costs examined here, as there are nearly no inexact MC paths with a relative error significantly away from 0 %. The few outliers are not serious. However, the memory consumption is quite large. But similar techniques as used for ATCHs (see Section 5.4) may reduce the memory consumption considerably.

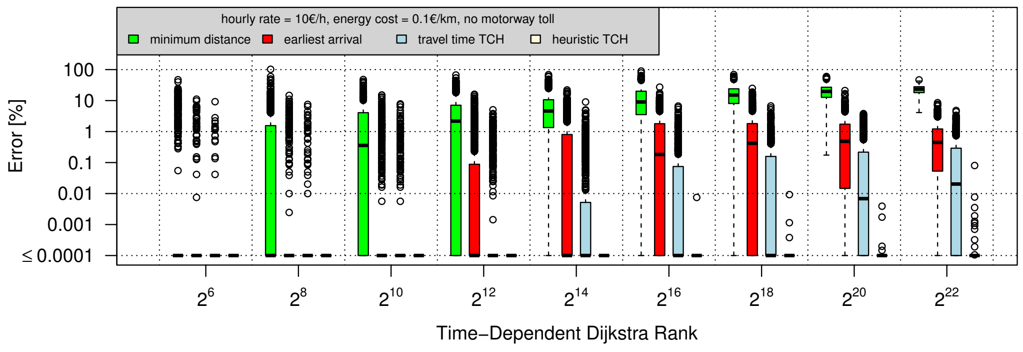
Please note that the implementation of the heuristic preprocessing (i.e., node ordering) is partly prototypical and considerably slower than the implementation of the TCH node ordering used in Section 5.6. Considering the node ordering time of the travel time TCHs, where the same implementation is used as for the node ordering of the heuristic TCHs, makes this obvious: There, the preprocessing does in principle the same work as the implementation used in Section 5.5 but takes more than five times longer (see Table 5.3 for comparison). It may hence be possible to do the heuristic preprocessing faster; for example, with a more cache efficient implementation of edges weights $f|c \in \mathscr{F}_{\Pi}^{1} \times \mathscr{X}_{\Pi}$. Nevertheless, ordering time and space usage are greater in case of heuristic TCHs than in case of travel time TCHs; especially, with motorway tolls. A possible explanation is that CPs have greater complexity than TTPs in theory; namely, up to $2^{\Omega(|V|)}$ bend points compared to at most $K \cdot |V|^{O(\log|V|)}$ bend points in case of TTPs, if $G = (V, E)$ has K bend points in total (see Section 6.1.2). But we have not analyzed this further.

An important result is that simply using travel time TCHs with the MC query procedure (see Algorithm 6.2) instead of the EA query procedure (see Algorithm 5.4) is not a good alternative to heuristic TCHs. Though without motorway tolls the average error is tolerable according to Table 6.3, there are several serious outliers that spoil the result. But in practice, even a few serious outliers may annoy some users, and this may already lead to a bad publicity. This can affect the success of a product or service. So, inexact MC queries with travel time TCHs are inappropriate. With motorway tolls, it is even worse as we see below.

Rank queries. Figure 6.6 and 6.7 show the distribution of the relative error over time-dependent Dijkstra rank for different kinds of heuristically obtained paths compared to exact MC paths computed with time-dependent multi-label A* search. Especially, Figure 6.7 is interesting. It shows, amongst others, the relative error of inexact MC queries on heuristic TCHs with enabled motorway tolls for time-dependent Dijkstra ranks up to 2^{20}; an information we have to omit in Table 6.3 because multi-label A* search does not terminate in reasonable time for random pairs of start and destination node with motorway tolls. It turns out that the error of heuristic MC queries is larger with motorway tolls than without, but still small enough to consider it as negligible from the practical point of view.

The error of inexact MC queries on travel time TCHs is also shown. Without motorway tolls there are serious outliers but the average error may be considered as tolerable. With motorway tolls it is different. With an hourly rate of 5 €/h, the





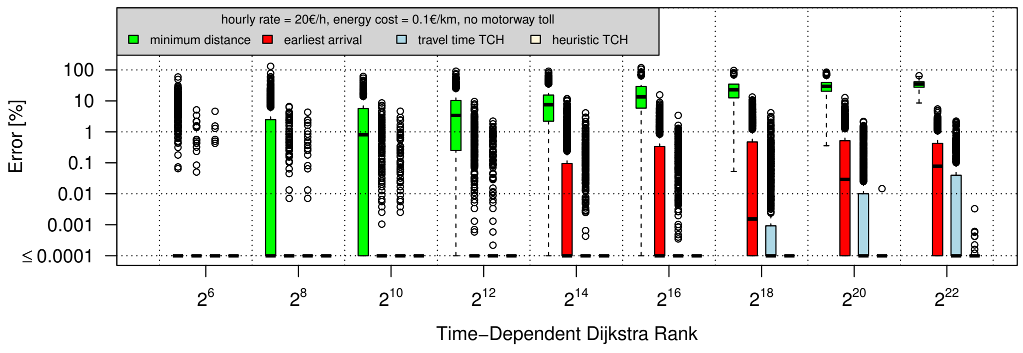


Figure 6.6. Relative error of different kinds of heuristically obtained paths compared to the minimum possible (i.e., optimal) travel cost for *Germany midweek* without motorway tolls, all plotted over time-dependent Dijkstra rank. Kinds of heuristic paths are minimum distance paths (green), EA paths (red), paths computed by inexact MC queries on travel time TCHs (blue), as well as by inexact MC queries on heuristic TCHs (yellow).
The number of paths is 1 000 per rank for each kind of heuristic paths. For every rank a separate distribution is represented as a box plot like in Figure 5.13. Less than 25 % of the paths computed by heuristic TCHs have relative error > 0.0001 %. So, all boxes and whiskers degenerate to a small bar at the bottom in this case. Only a few outliers can be seen.

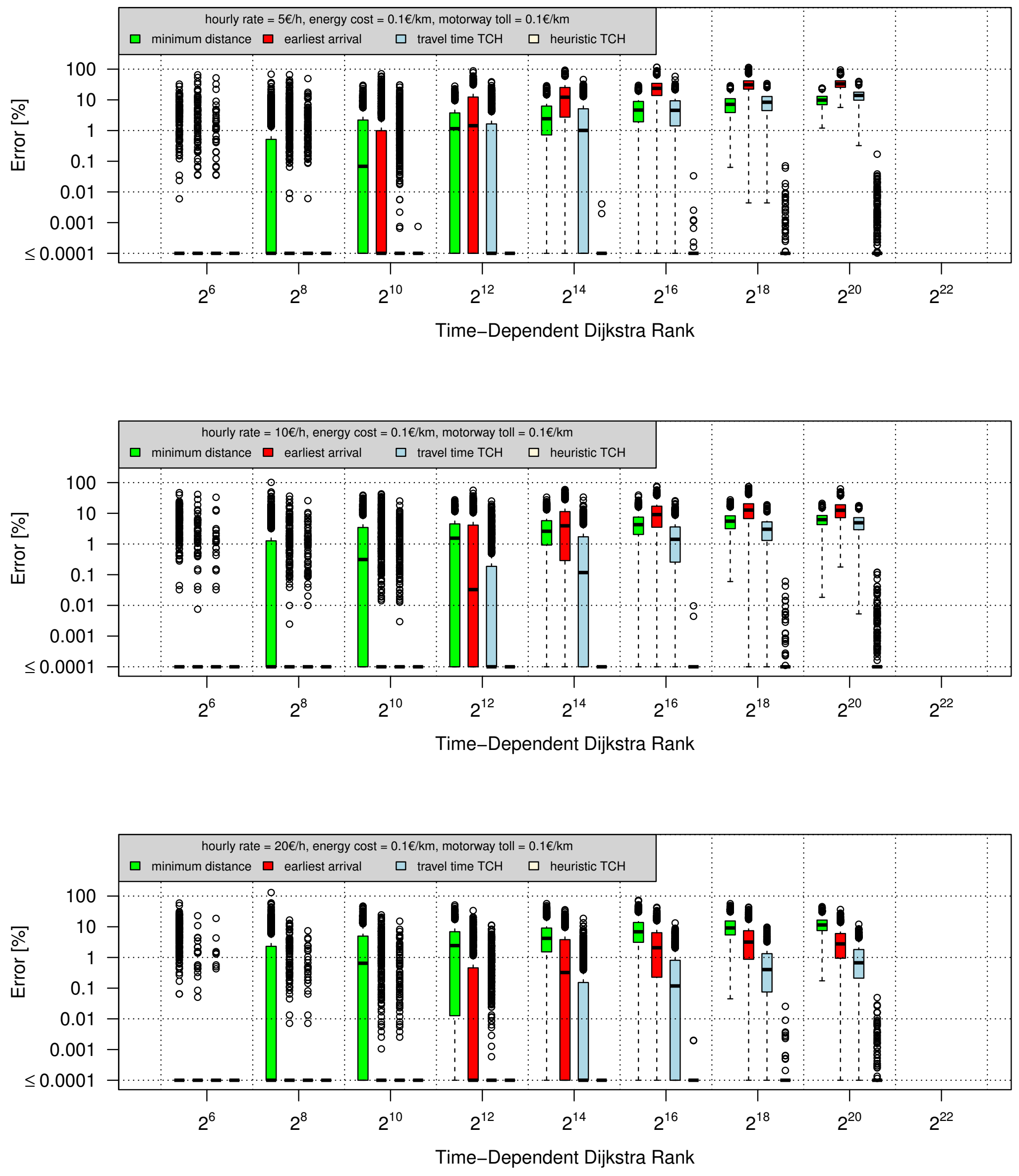


Figure 6.7. Relative error of different kinds of heuristically obtained paths compared to the minimum possible (i.e., optimal) travel cost for *Germany midweek* with motorway tolls of 0.1 €/km, all plotted over time-dependent Dijkstra rank. For rank 2^{22}, we do not report any relative errors because the exact algorithm (i.e., multi-label A* search) is not feasible there. The rest is as in Figure 6.6.

half of the routes has an error of about 1 % or more for time-dependent Dijkstra rank 2^{14}. For rank 2^{16}, more than 75 % of the routes have an error clearly above 1 %. This goes up to a median above 13 % in case of rank 2^{20}. For hourly rates of 10 €/h and 20 €/h the errors are smaller but still relatively large. It is obvious that inexact MC queries on travel time TCHs are not suited as heuristic method to compute MC paths.

Figure 6.6 and 6.7 also report the relative error of minimum distance paths and EA paths compared to MC paths. Note that EA paths are different from the paths computed by the MC query applied to travel time TCHs.[4] Obviously, the quality gets worse with increasing time-dependent Dijkstra rank. Without motorway tolls, the error of EA paths is barely tolerable with serious outliers. The quality of minimum distance paths is very bad. With motorway tolls, EA paths have bad quality, too. The quality of minimum distance paths is partly better and partly worse than of EA paths in this setup, but can never be considered as good. The interesting thing about EA paths and minimum distance paths, respectively, is that their error provides information of how correlated the travel cost is with the travel time and the travel distance for different problem instances. But neither minimum distance nor EA paths are suited as heuristic MC paths in the context of time-dependent road networks with additional time-invariant costs.

6.5 References

This chapter is heavily based on a conference article published together with Sanders [9]. Many wordings of this articles have been used or rephrased, although everything is explained in much more detail. New are the discussion why node contraction with additional costs is problematic as well as the discussion of possible conditions for exact node contraction (see Section 6.2.1). Also new are the discussion of the correctness of the minimum cost (MC) query (see Section 6.3.2) as well as the discussion of stall-on-demand in the context of heuristic TCHs (see Section 6.3.3).

[4]EA paths can be computed by EA query (see Algorithm 5.4) applied to travel time TCHs or by time-dependent Dijkstra (see Algorithm 4.1) applied to the original road network.

7

Discussion

7.1 Conclusion

This thesis generalizes contraction hierarchies (CHs) [44] to work with *time-dependent travel costs*. We consider two kinds of time-dependent travel costs in this thesis: *time dependent travel times* as well as time-dependent travel times with *additional time-invariant costs*.

Travel Time Only. We demonstrate that time-dependent route planning computations without additional time-invariant costs can be performed efficiently in time and space (see Chapter 5). For earliest arrival (EA) and travel time profile (TTP) queries, we provide fast, exact, and space efficient methods. On continental size road networks, exact EA queries take a few milliseconds, and exact TTP queries less than half a second. On an instance of the German road network this reduces to less than 35 msec. The underlying representation of the German road network instance, an approximate TCH (ATCH) in this case, needs less than 1 GiB space. All this is mainly achieved by the careful use of approximation that improves the efficiency without loosing exactness.

With running times bewtween 0.38 sec and 0.55 sec (depending on the memory usage of the underlying ATCH structure), TTP queries on continental size road networks are surely practical but cannot be regarded as instantaneous. So, it may be worthwhile to sacrifice some exactness to gain further speedup. Accepting errors below 1.4 %, we obtain an average running time of 0.1 sec. Note that no other algorithmic framework except TCH-based techniques is able to provide feasible answering of TTP queries on continental size road networks so far.

An important lesson learned is that route planning with time-dependent travel times is far from being a straightforward extension of route planning with constant travel costs. Several nontrivial algorithmic ingredients are necessary to make

TCHs efficient in time and space. Moreover, the most basic ingredients—namely, the linking and the minimum operation on TTFs—are relatively difficult to implement. This is due to possible rounding problems that usually not occur in the lower levels of a hierarchy; that is, they usually occur when the preprocessing is already running for a while. This makes debugging a rather time-consuming task.

With Additional Time-Invariant Costs. Route planning with additional time-invariant costs (see Chapter 6) only seems to be a slight extension of time-dependent travel times on the first glance. This, however, is not the case. Instead, minimum cost (MC) queries are NP-hard[1] and cost profiles (CPs) are significantly more complex than travel time profiles (TTPs) in the worst case. It is hence not surprising that our method for MC queries is only inexact; although our experiments show negligible errors. The time needed to compute an MC path is—with less than 35 msec for the hardest test instance based on our German road network—clearly good enough for practical applications. The preprocessing is feasible but takes relatively long—1 hour and 23 minutes with 8 parallel threads in shared memory. It must be noted, however, that our implementation of the heuristic preprocessing is partly prototypical.

What prevents us from exact MC querying is that we are not able to provide a feasible exact preprocessing procedure so far. With the heuristic TCH structures produced by our heuristic preprocessing, an optimal route may not be found. One main obstacle is to decide whether a shortcut edge can be omitted or not during node contraction. But even if we were able to decide this efficiently, we still would have to deal with the fact that parallel edges cannot be merged as simply as without additional costs. All these problems come from the fact that MC paths lack prefix-optimality in the presence of additional time-invariant costs.

Generalized Additional Costs. Time-dependent travel times with additional time-invariant costs (see Section 3.2) are modeled as pairs $f|c \in \mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$ where $\mathscr{F}_\Pi$ is essentially the set of continuous travel time functions (TTFs). It turns out that $\mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$ is not closed under concatenation and merging of the corresponding routes in the road network. There is, however, a superset $\mathscr{F}^1_\Pi \times \mathscr{X}_\Pi \supseteq \mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$ that is closed under these operations. There, $\mathscr{F}^1_\Pi$ is essentially the set of TTFs with points of discontinuity, and $\mathscr{X}_\Pi$ the set of piecewise constant functions. More precisely, TTFs are now allowed to jump, and additional costs are now time-dependent (but can only change by jumping). It is interesting that heuristic TCH structures (see Chapter 6) unavoidably contain edge weights from $\mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$, even if the underlying road network only has edge weights from $\mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$. This means, heuristic TCHs have to deal with edge weights from $\mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$ anyway.

[1]This follows from the NP-hardness of a very restricted special case (see Ahuja et al. [3]).

This suggests that heuristic TCHs may also be suited to work with more general time-dependent road networks that originally come with discontinuous TTFs and time-dependent (i.e., piecewise constant) additional costs. We further discuss that in Section 7.2.

Correctness and Algorithmic Ingredients. We not only perform experimental evaluations but also provide proofs of correctness for all TCH-based query algorithms in this thesis. All the query algorithms are assembled of several Dijkstra-like algorithms. We proof the correctness of the Dijkstra-like algorithms, too (see Chapter 4). There, it turns out that all these Dijkstra-like algorithms have similar structure, though they differ in some typical characteristics. So, several proofs of correctness follow similar patterns. Moreover, some of the Dijkstra-like algorithms are even special cases of others.

7.2 Future Work

Many aspects of time-dependent route planning are still unconsidered. Here, we discuss some questions that emerge naturally from the work described in this thesis. We discuss them separately for time-dependent route planning with and without additional costs.

Travel Time Only. Section 4.3.7 discusses the running time of approximate TTP search. There, we argue that the approximation of the bounds $\underline{F}[u]$ and $\overline{F}[u]$ should save more running time than the approximation of the travel time functions (TTFs) stored in the edges. This could be utilized to further speed up travel time profile (TTP) queries, both with ATCHs (see Section 5.4.3) and inexact TCHs (see Section 5.5.2). In case of inexact TCHs, one could also replace the conservative bounds by the minimum and maximum of the original TTFs to allow better pruning. For ATCHs it is similar. That means, additionally to the bounds $\underline{f}$ and $\overline{f}$ that replace the TTF f of an edge $u \rightarrow_f v$, edges could be annotated with $\min f$ and $\max f$. The improved pruning could not only speed up TTP queries but also EA queries, for both inexact TCHs and ATCHs.

Approximation may also help to make the preprocessing faster by further accelerating the witness search (see Section 5.2.2). Performing an approximate TTP search after the TTP interval search may prevent several expensive TTP searches during node contraction. Moreover, exact TTP search could be replaced by the much faster corridor contraction. Approximation could also be used to make the preprocessing more space efficient and more robust against rounding problems. The latter applies especially to the higher levels of the hierarchy. There, the bend points are the result of several linking and minimum operations and necessary shortcuts may mistakenly be omitted because of rounding problems. To overcome this, we could transfer the central idea of ATCHs to TCH preprocessing. That means, we no longer store exact TTFs in the shortcut edges but lower and an upper bounds. Only original edges have exact TTFs during this modified version of preprocessing. The gap between the lower and upper bounds makes it less likely that a necessary shortcut is omitted because of rounding errors. Of course, this may also reduce the running time and the space consumption of the preprocessing.

The cost term configuration of the node ordering (see Section 5.2.3) has never been explored systematically. It may be the case that better configurations exist than the one found by Vetter. Faster or more space efficient preprocessing may be possible this way. Sample search, which turns out to be not very effective in our experiments, might also be improved by using multiple departure times.

As suggested by Foschini et al. [36], one could restrict the slopes of TTFs to ∞, 0, and -1. Additionally, time could be restricted to $\mathbb{Z}$ instead of $\mathbb{R}$. The resulting very special TTFs are closed under linking and minimum. They could be used to model time-dependent road restrictions, which may be interesting for practitioners. Using ideas from pixel-based line drawing, one could even use this setup to represent complicated TTFs with a lossless compression efficiently.

Some time ago Delling et al. came up with customizable route planning [23] (for a short summary see Section 1.3.1 on page 33). This is most interesting for practice, because edge weights can be changed arbitrarily without running a new preprocessing. An update only takes moderate running time and the latest traffic data can be taken into account easily hence. So, a time-dependent version of customizable route planning would surely be interesting. The techniques developed in the context of ATCHs (see Section 5.4) may help to keep the memory consumption and update times moderate. This may also be the case for a time-dependent version of the recent customizable CHs [31] (for a short summary also see Section 1.3.1 on page 33).

Two problems that are also interesting from the application point of view are mobile time-dependent route planning and time-dependent turn costs. With respect to mobile time-dependent route planning, there are promising preliminary results provided by Kaufmann [53] in a bachelor thesis. Simulating a mobile device that computes nearly exact earliest arrival (EA) routes for the German road network, Kaufmann finds that 102 blocks of size 4 KiB are loaded from flash memory on average. This means query times below 0.15 sec should be possible if we assume that loading 4 KiB from flash memory takes 1.3 msec. Dividing the periodic time domain $[0,\Pi)$ into a number of intervals and then storing the TTFs as small sections grouped by these intervals may also reduce block loads and query time.

With Additional Costs. The two main obstacles of node contraction in the presence of additional costs (see Section 6.2.1) are

- to decide, whether a shortcut can be safely omitted or not, and
- to merge parallel shortcuts without loosing subpaths of minimum cost (MC) paths.

Regarding the first obstacle, this thesis discusses three conditions that are differently strong and presumably differently difficult to check. Designing practical procedures on basis of these conditions and evaluating their performance experimentally would be very interesting. Regarding the second obstacle, the underlying problem is to decide on which departure intervals a shortcut contributes to an MC path. So, similar ideas as regarding the first obstacle may be helpful. Then, merging of parallel shortcuts may be possible without annotating the shortcuts with too much overlapping partial TTFs (see Section 6.2.1 on page 278). This way, time-dependent route planning with additional time-invariant costs may become exact. That our heuristic TCHs show very small errors in our experiments (see Section 6.4), raises the hope that real-life instances may be manageable.

Regarding the question whether a shortcut can be safely omitted or not, it is an interesting option that the omitting of shortcuts may become obsolete in the not so far future. This is due to customizable CHs [31] mentioned before in this section. This variant of CHs has the great advantage that *full CH structures* are used; that is, no shortcut is omitted during node contraction. That the hierarchy does not get too dense, is because a completely different node order is used than in case of original CHs [44]. Of course, this does not solve the problem how parallel shortcut edges can be merged without, on the one hand, loosing subpaths of MC paths and, on the other hand, without generating too much overlapping partial TTFs.

In Section 7.1 we suggest that heuristic TCHs may be able to deal with the more general case that TTFs have points of discontinuity and that additional costs are piecewise constant (i.e., time-dependent). If an efficient exact node contraction procedure can be found, then this may even end up in a more general framework for exact time-dependent route planning with piecewise constant additional costs. There, it must be noted that points of discontinuity in time-dependent travel costs change the shortest path structure; that is, waiting or cycles (if waiting is forbidden) may become beneficial (see Section 3.2.4). The latter would require us to handle "loops" in the context of TCHs.[2] It must further be noted that TTFs violating the FIFO property make the shortest path structure of the original road network so inconvenient that the existence of Pareto prefix-optimal MC paths is

[2]Geisberger and Vetter [46] also deal with loops within a variant of constant travel cost CHs that supports turn costs.

no longer guaranteed. This even makes multi-label search (see Section 4.4.1) inapplicable to compute exact solutions.

However, if all TTFs fulfill the FIFO property (i.e., only positive discontinuities are allowed) and if waiting is allowed at the source nodes of edges with piecewise constant additional costs, then a heuristic (or even exact) TCH that is suited for time-invariant additional costs may also work with time-dependent additional costs. There, waiting can be simulated by TTFs with line segments of slope -1 (as suggested by Orda and Rom [69]). Note that the techniques developed in the context of ATCHs may be applicable to such TCHs, both in case of time-invariant and time-dependent additional costs. This way a lot of memory may be saved. An obvious application of piecewise constant (i.e., time-dependent) additional costs are time-dependent road charges. Or, time-dependent penalties could be introduced to keep the traffic away from hospitals or residential areas at night. But remember that piecewise constant additional costs introduce waiting or cycles to a time-dependent shortest path problem. This may be avoided by even more general additional time-dependent costs, beyond piecewise constant functions.

Another interesting issue are cost profile (CP) queries. Similar techniques as in case of ATCHs (see Section 5.4.3) could be used to obtain a small corridor where the expensive backward CP search is performed. However, the ideas of corridor contraction (see page 240) may not be fully applicable because MC paths lack prefix optimality, which makes merging of parallel shortcuts problematic. A possible solution may be to contract only nodes with exactly one incoming and one outgoing edge. Backward CP search is performed afterwards on the resulting contracted corridor.

7.3 Outlook

Route planning techniques that take changing traffic situations into account (i.e, time-dependent as well as dynamic techniques) all have the problem, that a critical number of users can change the traffic situation. Assume, for example, that several people use the same time-dependent route guidance system. If all these people are similarly rerouted to avoid the same regular congestion, then another congestion may arise on the suggested route. So, what we actually need, is a centralized time-dependent route guidance system that performs traffic simulation based on the current traffic situation every few minutes. The resulting time-dependent routes can not only take the current traffic situation into account, but also the effect on the future traffic situation of the time-dependent route guidance system itself.

To make the latest simulation results available for the route computation, a quickly updatable time-dependent route planning technique must be used. This could be a time-dependent version of customizable route planning [23] or of customizable CHs [31] for example. Both techniques only deal with constant travel costs so far. This emphasizes that time-dependent customizable route planning and customizable time-dependent CHs are an interesting topic. Time-dependent route planning with additional costs is also an issue in this context. For example, to keep users of centralized time-dependent route guidance systems away from residential areas or to penalize inconvenient roads.

Bibliography

[1] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks. In *International Symposium on Experimental Algorithms (SEA 2011)*, volume 6630 of *LNCS*, pages 230–241. Springer, 2011.

[2] Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. Highway Dimension, Shortest Paths, and Provably Efficient Algorithms. In *ACM-SIAM Symposium on Discrete Algorithms (SODA 2010)*, pages 782–793. SIAM, 2010.

[3] Ravindra K. Ahuja, James B. Orlin, Stefano Pallottino, and Maria G. Scutellà. Dynamic Shortest Paths Minimizing Travel Times and Costs. *Networks*, 41(4):197–205, 2003.

[4] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato Werneck. Route Planning in Transportation Networks. Technical Report MSR-TR-2014-4, Microsoft Research, 2014.

[5] Gernot Veit Batz, Daniel Delling, Peter Sanders, and Christian Vetter. Time-Dependent Contraction Hierarchies. In *Workshop on Algorithm Engineering and Experiments (ALENEX 2009)*, pages 97–105. SIAM, 2009.

[6] Gernot Veit Batz, Robert Geisberger, Sabine Neubauer, and Peter Sanders. Time-Dependent Contraction Hierarchies and Approximation. In *International Symposium on Experimental Algorithms (SEA 2010)*, volume 6049 of *LNCS*, pages 166–177. Springer, 2010.

[7] Gernot Veit Batz, Robert Geisberger, and Peter Sanders. Time Dependent Contraction Hierarchies – Basic Algorithmic Ideas. Technical report, Universität Karlsruhe (TH), 2008. arXiv:0804.3947.

[8] Gernot Veit Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. Minimum Time-Dependent Travel Times with Contraction Hierarchies. *ACM Journal of Experimental Algorithmics*, 18(1.4):1–43, 2013.

[9] Gernot Veit Batz and Peter Sanders. Time-Dependent Route Planning with Generalized Objective Functions. In *European Symposium on Algorithms (ESA 2012)*, volume 7501 of *LNCS*, pages 169–180. Springer, 2012.

[10] Reinhard Bauer and Daniel Delling. SHARC: Fast and Robust Unidirectional Routing. *ACM Journal of Experimental Algorithmics*, 14(2.4):1–29, 2009.

[11] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-directed Speed-up Techniques for Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 15(2.3):1–31, 2010.

[12] Annabell Berger, Martin Grimmer, and Matthias Müller-Hannemann. Fully Dynamic Speed-Up Techniques for Multi-criteria Shortest Path Searches in Time-Dependent Networks. In *International Symposium on Experimental Algorithms (SEA 2010)*, volume 6049 of *LNCS*, pages 35–46. Springer, 2010.

[13] Annabell Berger and Matthias Müller-Hannemann. Subpath-Optimality of Multi-Criteria Shortest Paths in Time- and Event-Dependent Networks. Technical report, Martin-Luther-Universität Halle-Wittenberg, 2009.

[14] Edith Brunel, Daniel Delling, Andreas Gemsa, and Dorothea Wagner. Space-Efficient SHARC-Routing. In *International Symposium on Experimental Algorithms (SEA 2010)*, volume 6049 of *LNCS*, pages 47–58. Springer, 2010.

[15] Tom Caldwell. On Finding Minimum Routes in a Network with Turn Penalties. *Communications of the ACM*, 4(2):107–108, 1961.

[16] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[17] Brian C. Dean. Shortest Paths in FIFO Time-Dependent Networks: Theory and Algorithms. Technical report, Massachusetts Institute Of Technology, 2004.

[18] Frank Dehne, Masoud T. Omran, and Jörg-Rüdiger Sack. Shortest Paths in Time-Dependent FIFO Networks. *Algorithmica*, 62(1-2):416–435, 2012.

[19] Daniel Delling. Time-Dependent SHARC-Routing. In *European Symposium on Algorithms (ESA 2008)*, volume 5193 of *LNCS*, pages 332–343. Springer, 2008.

[20] Daniel Delling. *Engineering and Augmenting Route Planning Algorithms*. Doctoral thesis, Universität Karlsruhe (TH), 2009.

[21] Daniel Delling. Time-Dependent SHARC-Routing. *Algorithmica*, 60(1):60–94, 2011.

[22] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. PHAST: Hardware-Accelerated Shortest Path Trees. *Journal of Parallel and Distributed Computing*, 73(7):940–952, 2013.

[23] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable Route Planning. In *International Symposium on Experimental Algorithms (SEA 2011)*, volume 6630 of *LNCS*, pages 376–387. Springer, 2011.

[24] Daniel Delling and Giacomo Nannicini. Core Routing on Dynamic Time-Dependent Road Networks. *INFORMS Journal on Computing*, 24(2):187–201, 2012.

[25] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering Route Planning Algorithms. In *Algorithmics of Large and Complex Networks*, volume 5515 of *LNCS*, pages 117–139. Springer, 2009.

[26] Daniel Delling and Dorothea Wagner. Landmark-Based Routing in Dynamic Graphs. In *International Workshop on Experimental Algorithms (WEA 2007)*, volume 4525 of *LNCS*, pages 52–65. Springer, 2007.

[27] Daniel Delling and Dorothea Wagner. Pareto Paths with SHARC. In *International Symposium on Experimental Algorithms (SEA 2009)*, volume 5526 of *LNCS*, pages 125–136. Springer, 2009.

[28] Daniel Delling and Dorothea Wagner. Time-Dependent Route Planning. In *Robust and Online Large-Scale Optimization*, volume 5868 of *LNCS*, pages 207–230. Springer, 2009.

[29] Daniel Delling and Renato F. Werneck. Faster Customization of Road Networks. In *International Symposium on Experimental Algorithms (SEA 2013)*, volume 7933 of *LNCS*, pages 30–42. Springer, 2013.

[30] Ugur Demiryurek, Farnoush Banaei-Kashani, and Cyrus Shahabi. A Case for Time-Dependent Shortest Path Computation in Spatial Networks. In *ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS 2010)*, pages 474–477. ACM, 2010.

[31] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable Contraction Hierarchies. In *International Symposium on Experimental Algorithms (SEA 2014)*, volume 8504 of *LNCS*, pages 271–282. Springer, 2014.

[32] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable Contraction Hierarchies. Technical report, Karlsruhe Institute of Technology, 2014. arXiv:1402.0402v4.

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

[34] David H. Douglas and Thomas K. Peucker. Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature. *The Canadian Cartographer*, 10(2):112–122, 1973.

[35] Stuart E. Dreyfus. An Appraisal of Some Shortest-Path Algorithms. *Operations Research*, 17(3):395–412, 1969.

[36] Luca Foschini, John Hershberger, and Subhash Suri. On the Complexity of Time-Dependent Shortest Paths. In *ACM-SIAM Symposium on Discrete Algorithms (SODA 2011)*, pages 327–341. SIAM, 2011.

[37] Stefan Funke and Sabine Storandt. Polynomial-time Construction of Contraction Hierarchies for Multi-criteria Objectives. In *Meeting on Algorithm Engineering and Experiments (ALENEX 2013)*, pages 41–54. SIAM, 2013.

[38] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[39] Robert Geisberger. Engineering Time-Dependent One-To-All Computation. Technical report, Karlsruhe Institute of Technology, 2010. arXiv:1010.0809.

[40] Robert Geisberger, Moritz Kobitzsch, and Peter Sanders. Route Planning with Flexible Objective Functions. In *Workshop on Algorithm Engineering and Experiments (ALENEX 2010)*, pages 124–137. SIAM, 2010.

[41] Robert Geisberger, Michael N. Rice, Peter Sanders, and Vassilis J. Tsotras. Route Planning with Flexible Edge Restrictions. *ACM Journal of Experimental Algorithmics*, 17(1.2):1–20, 2012.

[42] Robert Geisberger and Peter Sanders. Engineering Time-Dependent Many-to-Many Shortest Paths Computation. In *Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS 2010)*, volume 14 of *OASIcs*, pages 74–87. Schloss Dagstuhl, 2010.

[43] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *International Workshop on Experimental Algorithms (WEA 2008)*, volume 5038 of *LNCS*, pages 319–333. Springer, 2008.

[44] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Science*, 46(3):388–404, 2012.

[45] Robert Geisberger and Dennis Schieferdecker. Heuristic Contraction Hierarchies with Approximation Guarantee. In *Symposium on Combinatorial Search (SOCS 2010)*, pages 31–38. AAAI, 2010.

[46] Robert Geisberger and Christian Vetter. Efficient Routing in Road Networks with Turn Costs. In *International Symposium on Experimental Algorithms (SEA 2011)*, volume 6630 of *LNCS*, pages 100–111. Springer, 2011.

[47] Andrew V. Goldberg and Chris Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *ACM-SIAM Symposium on Discrete Algorithms (SODA 2005)*, pages 156–165. SIAM, 2005.

[48] Pierre Hansen. Bicriterion Path Problems. In *Multiple Criteria Decision Making Theory and Application*, volume 177 of *Lecture Notes in Economics and Mathematical Systems*, pages 109–127. Springer, 1980.

[49] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[50] Francis S. Hill, Jr. The Pleasures of "Perp Dot" Products. In *Graphics Gems IV*, pages 138–148. Academic Press, 1994.

[51] Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering Multilevel Overlay Graphs for Shortest-path Queries. *ACM Journal of Experimental Algorithmics*, 13(2.5):1–26, December 2009.

[52] Hiroshi Imai and Masao Iri. An Optimal Algorithm for Approximating a Piecewise Linear Function. *Journal of Information Processing*, 9(3):159–162, 1987.

[53] Harris Kaufmann. Towards Mobile Time-Dependent Route Planning. Bachelor thesis, Karlsruhe Institute of Technology, 2013.

[54] Tim Kieritz, Dennis Luxen, Peter Sanders, and Christian Vetter. Distributed Time-Dependent Contraction Hierarchies. In *International Symposium on Experimental Algorithms (SEA 2010)*, volume 6049 of *LNCS*, pages 83–93. Springer, 2010.

[55] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley, Pearson international edition, 2006.

[56] Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. Computing Many-to-Many Shortest Paths Using Highway Hierarchies. In *Workshop on Algorithm Engineering and Experiments (ALENEX 2007)*, pages 36–45. SIAM, 2007.

[57] Ulrich Lauther. An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. In *Geoinformation und Mobilität – von der Forschung zur praktischen Anwendung*, volume 22, pages 219–230. IfGI prints, 2004.

[58] Dennis Luxen and Christian Vetter. Real-Time Routing with OpenStreetMap Data. In *ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS 2011)*, pages 513–516. ACM, 2011.

[59] Lawrence Mandow and José Luis Pérez de la Cruz. Multiobjective A* Search with Consistent Heuristics. *Journal of the ACM*, 57(5.27):1–25, 2010.

[60] Ernesto Queirós Vieira Martins. On a Multicriteria Shortest Path Problem. *European Journal of Operational Research*, 16(2):236–245, 1984.

[61] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008.

[62] Nikola Milosavljević. On Optimal Preprocessing for Contraction Hierarchies. In *ACM SIGSPATIAL International Workshop on Computational Transportation Science (IWTCS 2012)*, pages 33–38. ACM, 2012.

[63] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning Graphs to Speed Up Dijkstra's Algorithm. In *International Workshop on Experimental and Efficient Algorithms (WEA 2005)*, volume 3503 of *LNCS*, pages 189–202. Springer, 2005.

[64] Giacomo Nannicini, Daniel Delling, Leo Liberti, and Dominik Schultes. Bidirectional A* Search for Time-Dependent Fast Paths. In *International Workshop on Experimental Algorithms (WEA 2008)*, volume 5038 of *LNCS*, pages 334–346. Springer, 2008.

[65] Giacomo Nannicini, Daniel Delling, Dominik Schultes, and Leo Liberti. Bidirectional A* Search on Time-Dependent Road Networks. *Networks*, 59(2):240–251, 2012.

[66] Sabine Neubauer. Space Efficient Approximation of Piecewise Linear Functions. Student research project (Studienarbeit), Universität Karlsruhe (TH), 2009.

[67] Tatsuya Ohshima, Pipaporn Eumthurapojn, Liang Zhao, and Hiroshi Nagamochi. An A* Algorithm Framework for the Point-to-Point Time-Dependent Shortest Path Problem. In *China-Japan Joint conference on Computational Geometry, Graphs and Applications (CGGA 2010)*, volume 7033 of *LNCS*, pages 154–163. Springer, 2011.

[68] Ariel Orda and Raphael Rom. Traveling Without Waiting in Time-Dependent Networks is NP-Hard. Manuscript, Dept. Electrical Engineering, Technion – Israel Institute of Technology, Haifa, Israel, 1989.

[69] 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.

[70] Ariel Orda and Raphael Rom. Minimum Weight Paths in Time-Dependent Networks. *Networks*, 21(3):295–319, 1991.

[71] PTV Planung Transport Verkehr AG. http://www.ptvgroup.com, 1979.

[72] Urs Ramer. An Iterative Procedure for the Polygonal Approximation of Plane Curves. *Computer Graphics and Image Processing*, 1(3):244–256, 1972.

[73] Peter Sanders. Personal communication, 2012.

[74] Peter Sanders and Dominik Schultes. Dynamic Highway-Node Routing. In *International Workshop on Experimental Algorithms (WEA 2007)*, volume 4525 of *LNCS*, pages 66–79. Springer, 2007.

[75] Peter Sanders and Dominik Schultes. Engineering Highway Hierarchies. *ACM Journal of Experimental Algorithmics*, 17(1.6):1–40, 2012.

[76] Peter Sanders, Dominik Schultes, and Christian Vetter. Mobile Route Planning. In *European Symposium on Algorithms (ESA 2008)*, volume 5193 of *LNCS*, pages 732–743. Springer, 2008.

[77] Dennis Schieferdecker. Systematic Combination of Speed-Up Techniques for exact Shortest-Path Queries. Diploma thesis, Universität Karlsruhe (TH), 2008.

[78] Dominik Schultes. *Route Planning in Road Networks*. Doctoral thesis, Universität Karlsruhe (TH), 2008.

[79] Robert Sedgewick. *Algorithms*. Addison-Wesley, second edition, 1988.

[80] Christian Sommer. Shortest-path Queries in Static Networks. *ACM Computing Surveys*, 46(4.45):1–31, 2014.

[81] Sabine Storandt. Route Planning for Bicycles – Exact Constrained Shortest Paths Made Practical via Contraction Hierarchy. In *International Conference on Automated Planning and Scheduling*, pages 234–242. AAAI, 2012.

[82] Christian Vetter. Parallel Time-Dependent Contraction Hierarchies. Student research project (Studienarbeit), Universität Karlsruhe (TH), 2009.

[83] Dorothea Wagner. Algorithm Engineering for Route Planning – An Update. In *International Symposium on Algorithms and Computation (ISAAC 2011)*, volume 7074 of *LNCS*, pages 1–5. Springer, 2011.

[84] Wolfgang Walter. *Analysis 1*. Springer, third edition, 1992. In German.

[85] Stephan Winter. Modeling Costs of Turns in Route Planning. *GeoInformatica*, 6(4):345–361, 2002.

List of Notations

$\prec$	—	node order	68
$\preceq$	—	total order	58
$\mathbf{arr}\, f$	—	arrival time function of TTF f	79
$\lvert c \rvert$	—	complexity of p.w.c. function c	104
$ccw(p,q,r)$	—	orientation of points $p,q,r \in \mathbb{R}^2$ in the plain	56
$c \vert d$	—	two-dimensional edge cost	54
$\mathrm{Cost}_G(s,t,\tau_0)$	—	minimum travel cost from s to t with departure time τ_0	99
$\overline{\mathrm{Cost}}_G(s,t)$	—	upper bound of minimum travel cost from s to t	100
$\underline{\mathrm{Cost}}_G(s,t)$	—	lower bound of minimum travel cost from s to t	100
$\overline{C}_P$	—	upper bound of total cost of path P	98
$\underline{C}_P$	—	lower bound of total cost of path P	98
C_P	—	TCF of path P (with constant additional costs)	98
C_P	—	TCF of path P (with p.w.c. additional costs)	107
c_P	—	total constant cost of path P	53
c_P	—	total additional time-invariant cost of path P	98
c_P	—	p.w.c. (i.e., time-dependent) ACF of path P	107
$c_P \vert d_P$	—	two-dimensional total cost of a path P	54

$(\underline{C}[u], \overline{C}[u])$	—	pair of TCF bounds as label of node u	159
Δ	—	current search direction $\Delta \in \{s,t\}$	61
$\mathbf{dep}\, f$	—	departure time function of TTF f	81
$d *_f c$	—	linking $c, d \in \mathscr{X}_\Pi$ with respect to TTF $f \in \mathscr{F}_\Pi^1$	105
$\mathrm{dom}\, f$	—	domain of function f	51
$d[u]$	—	constant travel cost as label of node u	59
E	—	set of edges of a directed graph G	52
$\mathrm{EA}_G(s,t,\tau_0)$	—	EA time from s at t in G with departure time τ_0	87
E_H	—	edge set of TCH structure	187
$\lvert f \rvert$	—	complexity of a periodic p.w.l. function	78
f^{-1}	—	inverse function of f	50
$\underline{f}$	—	lower bound of TTF f	128
$\overline{f}$	—	upper bound of TTF f	128
$\tilde{f}$	—	inexact version of TTF f	242
$f \vert c$	—	pair of TTF $f \in \mathscr{F}_\Pi^1$ and p.w.c. ACF $c \in \mathscr{X}_\Pi$	104
$f + g$	—	pointwise sum of real functions	50
f_i	—	i-th bend point of a p.w.l. function	79
$f_i.x$	—	x-value of i-th bend point of a p.w.l. function	79
$f_i.y$	—	y-value of i-th bend point of a p.w.l. function	79
f_P	—	TTF of path P (continuous TTFs)	87
f_P	—	TTF of path P (TTFs with discontinuities)	107
$\mathscr{F}_\Pi$	—	set of continuous periodic TTFs with FIFO property	79
$\mathscr{F}_\Pi^+$	—	subset of $\mathscr{F}_\Pi$ with strengthened FIFO property	81
$\mathscr{F}_\Pi^1$	—	extension of $\mathscr{F}_\Pi$ to left-continuous TTFs	102

$F[u]$	— TTF as label of node u	123
$F[u]\|C[u]$	— TTF and ACF as label of node u	150
$(\underline{F}[u], \overline{F}[u])$	— pair of TTF bounds as label of node u	128
$\|f\|$	— Number of bend point of p.w.l. function f	78
G	— directed graph	52
$G^\top$	— transpose graph	52
$g\|d \star f\|c$	— linking of pairs $f\|c,\ g\|d \in \mathscr{F}_\Pi^1 \times \mathscr{X}_\Pi$	105
$g * f$	— Linking of TTFs f, g	82
$g \circ f$	— composition of real functions	50
$G(L)$	— predecessor graph (multi-label search)	118
$G(p)$	— predecessor graph (single-label search)	118
H	— TCH structure	187
$H_\downarrow$	— downward graph of a CH or TCH	72
$H_\uparrow$	— upward graph of a CH or TCH	72
$\tilde{H}$	— inexact TCH structure	243
id	— identity function	50
$(i, u, \gamma\|\delta, i')$	— label of a node u (multi-label search)	64
ℓ	— label of a node (multi-label search)	64
$\mathrm{LD}_G(s, t, \sigma_0)$	— LD time from s at t in G with arrival time σ_0	90
$\lim_{x \to x_0^-} f(x)$	— left-sided limit	51
$\lim_{x \to x_0^+} f(x)$	— right-sided limit	51
$L[u]$	— label set of a node u (multi-label search)	64
$\mathrm{MCTT}_G(s, t, \tau_0)$	— minimum travel time of an (s, t, τ_0)-MC-path	101
$\min_{c,d}(f, g)$	— merging $f, g \in \mathscr{F}_\Pi^1$ with respect to $c, d \in \mathscr{X}_\Pi$	108

$\min(f\|c,\, g\|d)$	— merging $f\|c,\, g\|d \in \mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$	109
$\min(f,g)$	— pointwise minimum of real functions	50
$\min^{f,g}(c,d)$	— merging $c,d \in \mathscr{X}_\Pi$ with respect to $f,g \in \mathscr{F}^1_\Pi$	108
$\mu_G(s,t)$	— minimum (constant) travel cost from s to t in G	53
$N^k_G(u)$	— k-hop neighborhood of node u	52
Π	— period of TTFs, ACFs and TCFs	78
PP'	— concatenation of the paths P and P'	52
(pq)	— line segment between points $p,q \in \mathbb{R}^2$ excluding p,q	55
$p \bullet q$	— perp dot product of points $p,q \in \mathbb{R}^2$	55
π_t	— potential function (A* search)	63
$p[u]$	— predecessor information of node u	59
$\mathscr{P}_U(S,T)$	— united graph of all $\mathscr{P}_U(s,t)$ with $s \in S$ and $t \in T$	52
$\mathscr{P}_U(s,t)$	— graph formed by all paths from s to t in $U \subseteq G$	52
Q	— priority queue (PQ)	58
$[q[u], r[u]]$	— interval as node label	126
R	— remaining graph (TCH construction)	190
s	— start node (one-to-one query)	60
τ	— point in time	17
t	— destination node (one-to-one query)	60
$\textbf{tcf}\, f\|c$	— TCF represented by TTF f and ACF c	104
$\mathrm{TT}_G(s,t,\tau_0)$	— minimum travel time from s to t with departure time τ_0	90
$\overline{\mathrm{TT}}_G(s,t)$	— upper bound of minimum travel time from s to t	90
$\underline{\mathrm{TT}}_G(s,t)$	— lower bound of minimum travel time from s to t	90
$\tau[u]$	— point in time as label of node u	119
$u \to v$	— edge of a directed graph	52

$\langle u \to \cdots \to v \rangle$	—	path from node u to node v 52
$u \to_c v$	—	directed edge with non-negative constant travel cost c ... 52
$u \to_{c\|d} v$	—	directed edge with two-dimensional cost $c\|d$ 54
$u \to_{f\|C} v$	—	directed edge with TTF f and TCF C 77
$u \to_{f\|c} v$	—	directed edge with TTF f and TCF $f+c$ 97
$u \to_f v$	—	directed edge $u \to_{f\|C} v$ with TTF f and TCF $C := f$ 78
$u \to_{(\underline{f},\overline{f})} v$	—	edge of ATCH with bounds $\underline{f}$ and $\overline{f}$ 228
V	—	set of nodes of a directed graph G 52
(V,E)	—	directed graph $G = (V,E)$ 52
X	—	set of candidate nodes 211
$\mathscr{X}_\Pi$	—	set of p.w.c. left-continuous functions with period Π ... 104

Deutsche Zusammenfassung

Die rechnergestützte Routenplanung hat in den letzten zehn Jahren zunehmend an Relevanz gewonnen (siehe Überblicksartikel [4, 25, 28, 83]). Vor allem im Hinblick auf Straßennetze ist dieses Thema zu einer echten Erfolgsgeschichte geworden. Die schnellsten aktuellen Algorithmen sind sogar in der Lage, eine optimale Route im europäischen Straßennetz innerhalb von Mikrosekunden zu berechnen. Dijkstra's Algorithmus [33], vor ungefähr zehn Jahren noch das Standardverfahren zur Routenberechnung, würde dafür einige Sekunden benötigen. Durch diesen Fortschritt ist die rechnergestützte Routenplanung inzwischen zu einem selbstverständlichen Teil unseres Alltagslebens geworden. Populäre Webanwendungen wie Bing Maps, Google Maps oder HERE sind nur drei Beispiele.

Zeitabhängige Routenplanung. Die meisten Arbeiten im Bereich der rechnergestützten Routenplanung für Straßennetze modellieren Reisekosten nur auf sehr einfache Weise als konstante nummerische nicht-negative Werte, die typischerweise stark mit der Reisezeit korreliert sind. Manche praktische Aspekte werden so jedoch außer Acht gelassen. Dabei handelt es sich um periodisch wiederkehrende Phänomene wie Stoßzeiten oder regelmäßige Staus. In dieser Arbeit werden zwei Arten von zeitabhängigen Reisekosten betrachtet: *zeitabhängige Reisezeiten* und zeitabhängige Reisezeiten mit *zusätzlichen zeitunabhängigen Kosten*. Die Reisezeitinformationen müssen dabei die bekannte FIFO Eigenschaft erfüllen, was bedeutet, dass ein späterer Reisebeginn nie eine frühere Ankunft zur Folge hat.

Eine typische Anwendung für Routenplanung mit zeitabhängigen Reisezeiten ist das Vermeiden der bereits erwähnten Stoßzeiten und regelmäßig wiederkehrenden Staus (die zugrunde liegenden Reisezeitinformationen werden normalerweise aus statistischen Daten gewonnen). Die zusätzlichen zeitunabhängigen Kosten scheinen auf den ersten Blick nur eine leichte Erweiterung darzustellen. Tatsächlich wird die Routenberechnung durch diese aber deutlich erschwert. Das

Finden einer optimalen Route für eine gegebene Abfahrtszeit wird so nämlich zu einem NP-schweren Problem.[3]

Aus Anwendungssicht können die zusätzlichen zeitunabhängigen Kosten zum penalisieren ungünstiger Straßenabschnitte verwendet werden, etwa wenn Straßen sehr eng, reich an engen Kurven oder zu steil sind, oder wenn sie Gefahrenstellen mit erhöhtem Risiko für Glatteis oder Steinschlag aufweisen. Eine weitere Anwendungsmöglichkeit sind Reisekosten, die in Form von Geldbeträgen anfallen und sich aus einem zeitabhängigen und einem zeitunabhängigen Anteil zusammensetzen, wobei der zeitabhängige Anteil proportional zur Reisedauer sein muss. Ein Beispiel hierfür wären etwa Reisekosten, die einerseits durch den Stundenlohn eines LKW-Fahrers und andererseits durch zusätzliche Mautkosten auf Autobahnen entstehen.

Zeitabhängige Contraction Hierarchies. Ziel dieser Arbeit ist es, *Contraction Hierarchies (CHs)* [44] – eine erfolgreiche algorithmische Technik für schnelles und exaktes Berechnen von Routen in Straßennetzen – derartig zu verallgemeinern, dass sie auch mit zeitabhängigen Reisekosten funktionieren. Wie die meisten anderen Techniken im Bereich der Routenplanung auch, ermöglichen CHs ein sehr schnelles Beantworten von Nutzeranfragen, erfordern aber eine zeitaufwändige Vorverarbeitung.

Für den Fall, dass keine zusätzlichen zeitabhängigen Kosten vorliegen, sind die in dieser Arbeit behandelten Verfahren tatsächlich in der Lage, eine schnelle und exakte Berechnung von Routen zu gewährleisten (siehe Kapitel 5). Die zugrunde liegenden *time-dependent Contraction Hierarchies (TCHs)* – sowie deren Varianten *approximative TCHs (ATCHs)* und *nicht-exakte TCHs (engl. inexact TCHs)* – sind jedoch weit mehr als eine offensichtliche Erweiterung der ursprünglichen CHs. Stattdessen kommen ausgeklügelte algorithmische Techniken zum Einsatz. Das wichtigste Konzept ist dabei der geschickte Einsatz von approximativen Berechnungen, um kleine Teilgraphen des Straßennetzes zu extrahieren, die alle für die Berechnung relevanten Knoten und Kanten enthalten. Diese einfache aber effektive Idee ermöglicht es,

- exakte aber langsame Algorithmen schneller durchzuführen, ohne das Ergebnis der Berechnung zu verfälschen, und
- optimale Routen zu berechnen, obwohl ein erheblicher Teil der vorhanden Reisezeitinformationen lediglich approximiert vorliegt.

Der zweite Punkt ermöglicht zudem eine deutlich erhöhte Speichereffizienz. Für eine Instanz des deutschen Straßennetzes kann die minimal mögliche Reisezeit

[3]Dass die Routenberechnung mit zusätzlichen zeitabhängigen Kosten NP-schwer ist, ergibt sich daraus, dass bereits ein stark eingeschränkter Spezialfall dieses Problems NP-schwer ist [3].

für eine gegebene Abfahrtszeit z.B. in weniger als 1.2 msec berechnet werden. Ein sogenanntes *Reisezeitprofil* – also eine Funktion, die zu einer Abfahrtszeit die minimal mögliche Reisezeit liefert – kann in weniger als 35 msec berechnet werden. Gibt man sich dabei mit einen Fehler von ungefähr 1 % zufrieden, sinkt die Rechenzeit sogar auf 2.6 msec. Die zugrunde liegenden ATCHs und nicht-exakten TCHs, die in diesem Fall das Straßennetz repräsentieren, benötigen dabei weniger als 1 GiB Hauptspeicher. Die Vorberechnung zur Erzeugung dieser Strukturen benötigt etwa 30 Minuten. Mit 8 parallelen Verarbeitungsfäden und gemeinsamen Hauptspeicher verringert sich die Vorberechnungszeit auf ca. 5 Minuten.

Sind zusätzliche zeitunabhängige Kosten vorhanden, gelingt mit den Verfahren, die in dieser Arbeit vorgestellt werden, leider nur eine nicht-exakte Routenberechnung (siehe Kapitel 6). Die zugrunde liegenden *heuristischen TCHs* sind nämlich nicht in der Lage, für jede mögliche Kombination aus Start, Ziel und Abfahrtszeit das Finden einer optimalen Route zu ermöglichen. In unseren Experimenten bleibt der Fehler jedoch vernachlässigbar klein. Das finden einer möglichst kostengünstigen Route dauert für unsere schwierigste Testinstanz durchschnittlich unter 35 msec. Dabei handelt es sich wieder um das deutsche Straßennetz, wobei folgendes Kostenmodell zugrunde gelegt wird: Reisekosten fallen in Form von Geldbeträgen an. Der zeitabhängige Anteil ergibt sich dabei aus einem festgelegten Stundensatz (z.B. dem Stundenlohn eines Fahrers). Der zeitunabhängige Anteil setzt sich aus einer sehr einfachen Abschätzung von Energiekosten und einer Autobahnmaut zusammen. Sowohl Energiekosten als auch Maut werden als proportional zur gefahrenen Stecke angenommen, im Fall der Maut werden aber nur Autobahnkilometer gezählt.

Mit diesem Kostenmodell dauert die Vorberechnung 1 Stunde und 23 Minuten, bei 8 parallelen Verarbeitungsfäden in gemeinsamem Speicher. Dies ist deutlich länger als die 5 Minuten, die ohne zeitunabhängige Zusatzkosten benötigt werden. Allerdings wird für die Vorberechnung mit zeitunabhängigen Zusatzkosten eine prototypische Implementierung verwendet. Womöglich kann diese Laufzeit also noch verringert werden. Allerdings ist auch dann noch eine deutlich höhere Rechenzeit als ohne zeitunabhängige Zusatzkosten zu erwarten. Der Speicherbedarf der heuristischen TCH ist mit 9 GiB etwa doppelt so hoch wie ohne Zusatzkosten. Man beachte dabei, dass sich der weiter vorne angegebene Speicherbedarf von unter 1 GiB auf ATCHs bezieht, einer Variante von TCHs die durch Verwendung approximierter Reisezeitdaten eine erhebliche Speicherersparnis erreicht. Womöglich können solche Techniken auch mit zusätzlichen zeitunabhängigen Kosten verwendet werden, was bisher jedoch nicht ausprobiert worden ist. Das Hauptergebnis der Experimente mit zusätzlichen zeitunabhängigen Kosten ist, dass heuristische TCHs nahezu exakte Routenberechnung erlauben, bei praktikablen Laufzeiten bzgl. Vorberechnung und Routenberechnung.

Die Berechnung von Reisekostenprofilen wird in dieser Arbeit lediglich im

Rahmen der Vorberechnung betrachtet. Allerdings können wir zeigen, dass zeitunabhängige Zusatzkosten die Komplexität von Reisekostenprofile deutlich erhöhen (siehe Abschnitt 6.1.2). Demnach können Reisekostenprofile, dargestellt als stückweise lineare Funktionen, aus bis zu $2^{\Omega(n)}$ Liniensegmenten bestehen; dabei sei n die Anzahl der Knoten im Straßennetz. Dagegen können Reisezeitprofile – nach Foschini et al. [36] – höchstens $K \cdot n^{O(\log n)}$ Liniensegmente enthalten; K sei dabei die Gesamtzahl der Liniensegmente aller Reisezeitfunktionen im Straßennetz.

Verallgemeinerte zusätzliche Kosten. Zeitabhängige Reisezeiten mit zusätzlichen zeitunabhängigen Kosten werden üblicherweise als Paare $f|c \in \mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$ dargestellt, wobei $\mathscr{F}_\Pi$ die Menge der stückweise linearen stetigen Reisezeitfunktionen ist. Wie sich herausstellt, ist $\mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$ nicht abgeschlossen unter Konkatenation und Mischen der zugehörigen Routen. Allerdings existiert eine Obermenge $\mathscr{F}^1_\Pi \times \mathscr{X}_\Pi \supseteq \mathscr{F}_\Pi \times \mathbb{R}_{\geq 0}$ auf die dies zutrifft. Dabei ist $\mathscr{F}^1_\Pi$ die Menge der stückweise linearen Reisezeitfunktionen mit Unstetigkeitsstellen und $\mathscr{X}_\Pi$ die Menge der *stückweise konstanten* Funktionen. Genauer gesagt, können Reisezeitfunktionen nun Sprungstellen enthalten, und aus zeitunabhängigen zusätzlichen Kosten werden zeitabhängige zusätzliche Kosten, die ihren Wert jedoch nicht stetig sondern ausschließlich an Sprungstellen verändern. Interessant ist hierbei, dass die Kantengewichte heuristischer TCHs zwangsläufig aus $\mathscr{F}^1_\Pi \times \mathscr{X}_\Pi$ stammen, und zwar auch dann, wenn das ursprüngliche Straßennetz nur stetige Reisezeitfunktionen und zeitunabhängige Zusatzkosten aufweist (siehe Kapitel 6). Womöglich bedeutet dies, dass heuristische TCHs auch für Straßennetze mit zeitabhängigen (d.h. stückweise konstanten) Zusatzkosten geeignet sind. Zeitabhängige Mautkosten oder zeitabhängige Penalisierung von Straßenabschnitten wären mögliche Anwendungsfälle. Letzteres könnte z.B. helfen, nächtlichen Verkehr von Wohngebieten und Krankenhäusern fernzuhalten.

Korrektheitsbeweise. Für alle Algorithmen zur Routenberechnung, die in dieser Arbeit beschrieben werden, erfolgt auch ein Beweis ihrer Korrektheit. All diese Algorithmen sind weitgehend aus "Dijkstra-ähnlichen" Grundalgorithmen aufgebaut. In Kapitel 4 erfolgt eine eingehende Betrachtung all dieser Grundalgorithmen, wobei auch deren Korrektheit bewiesen wird. Zu beachten ist dabei, dass alle diese Grundalgorithmen eine ähnliche Grundstruktur aufweisen. Zudem sind manche dieser Grundalgorithmen auch Spezialfälle von anderen Grundalgorithmen. Aus diesem Grund werden einige der Korrektheitsbeweise nicht im Detail dargestellt. Stattdessen wird auf einen analogen Beweis oder einen jeweils allgemeineren Grundalgorithmus verwiesen.

Referenzen. Viele der Ergebnisse, die in dieser Arbeit beschrieben werden, sind bereits veröffentlicht; zum einen auf Konferenzen [5, 6, 9], zum anderen in einer Fachzeitschrift [8]. Außerdem gibt es einen frühen technischen Bericht [7]. Die Veröffentlichungen erfolgten zusammen mit Daniel Delling, Robert Geisberger, Sabine Neubauer, Peter Sanders, und Christian Vetter, in jeweils in unterschiedlichen Kombinationen. Aus den genannten Artikel wurden viele Formulierungen in diese Arbeit übernommen und dabei z.T. verändert.

An dieser Stelle ist es auch wichtig zu bemerken, dass Christian Vetter wichtige Beiträge zu den, in dieser Arbeit dargestellten, TCHs geleistet hat. Dabei ist ausschließlich die Vorberechnung betroffen. Zum einen hat Vetter im Rahmen seiner Studienarbeit [82] die Vorberechnung parallelisiert, zum anderen hat er als studentische Hilfskraft weitere Arbeiten an der Vorberechnung durchgeführt. Auf diese Weise hat er sehr stark zur Reduzierung der Vorberechnungszeit beigetragen. Insgesamt hat Vetter wesentlichen Anteil an den Ideen und der Implementierung der Vorberechnung (für Details siehe Abschnitt 5.7), weswegen er auch Mitautor zweier Artikel [5, 8] ist.

Dank gilt auch Sabine Neubauer, die uns ihre Implementierung des Imai-Iri-Algorithmus zum Approximieren von stückweise linearen Funktionen [52] zur Verfügung gestellt hat. Diese wurde von ihr im Rahmen ihrer Studienarbeit angefertigt [66]. Ohne diese Implementierung hätten ATCHs, nicht-exakte TCHs, sowie die schnelle Berechnung von Reisezeitprofilen nur mit zusätzlichem Arbeitsaufwand umgesetzt werden können.