Advanced Route Planning in Transportation Networks

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

von der Fakultät für Informatik
des Karlsruher Instituts für Technologie

genehmigte

Dissertation

von

Robert Geisberger

aus München

Tag der mündlichen Prüfung:	4. Februar 2011
Erster Gutachter:	Prof. Dr. Peter Sanders
Zweite Gutachterin:	Prof. Dr. Hannah Bast

Abstract

Many interesting route planning problems can be solved by computing shortest paths in a suitably modeled, weighted graph representing a transportation network. Such networks are naturally road networks or timetable networks of public transportation. For large networks, the classical Dijkstra algorithm to compute shortest paths is too slow. And therefore have faster algorithms been developed in recent years. These new algorithms have in common that they use precomputation and store auxiliary data to speed-up subsequent shortest-path queries. However, these algorithms often consider only the simplest problem of computing a shortest path between source and target node in a graph with non-negative real edge weights. But real-life problems are often not that simple, and therefore, we need to extend the problem definition. For example, we need to consider time-dependent edge weights, road restrictions, or multiple source and target nodes. While some of the previous algorithms can be used to solve such extended problems, they are usually inefficient. It is therefore important to develop new algorithmic ingredients, or even completely new algorithmic ideas. We contribute solutions to three practically relevant classes of such extended problems: *public transit networks*, *flexible queries*, and *batched shortest paths computation*. These classes are introduced in the following paragraphs.

Public transit networks, for example train and bus networks, are inherently event-based, so that a time-dependent model of the network is mandatory. Furthermore, as they are much less hierarchically structured than road networks, algorithms for road networks perform much worse or even fail, depending on the scenario. We contribute to both cases, first an algorithm that has its initial ideas taken from a technique for road networks, but with significant algorithmic changes. It is based on the concept of node contraction, but now applied to stations. A key point to the success was the usage of a station graph model, having a single node per station. In contrast, the established time-expanded or time-dependent models require multiple nodes per station in general. And second, we contribute to a new algorithm specifically designed for transit networks supporting a fully realistic scenario. It is based on the concept of transfer patterns: a connection is described as the sequence of station where a transfer happens. Although there are potentially hundreds of optimal connections between a pair of stations during a week, the set of transfer patterns is usually much smaller. The main problem is the computation of the transfer patterns. A naïve way would be a Dijkstra-like one-to-all query from each station. But this is too time-consuming so that we develop heuristics to achieve a feasible precomputation time.

The term *flexible queries* refers to the problem of computing shortest paths with further query parameters, such as the edge weight (e. g. travel time or distance), or restrictions (e. g. no toll roads). While Dijkstra's algorithm only requires small changes to support such query parameters efficiently, existing algorithms based on precomputation become much more complicated. A naïve adaption of these algorithms would perform a separate precomputation for each possible value of the query parameters, but this is

highly inefficient. We therefore design algorithms with a single joint precomputation. While it is easy to achieve a better performance than the naïve approach, it is a great algorithmic challenge to be almost as good as the previous algorithms that do not support flexibility. Also, depending on the query parameters, only parts of the precomputed data are interesting, and we need effective algorithms to prune the uninteresting data. Another obstacle is the classification of roads and junctions. For example, fast highways are very important for the travel time metric, but less so for distance metric. The so called "hierarchy" of the network changes with different values of the query parameters. This highly affects the efficiency of the previous algorithms and we develop new techniques to cope with that.

Batched shortest paths computation comprises problems considering multiple source-target pairs. The classical problem is the computation of a shortest paths distance table between a set of source nodes S and a set of target nodes T. Of course, any of the previous algorithms can compute this table. But much faster algorithms are possible, which exploit that we want to compute the distances between *all pairs* in $S \times T$. Such tables are important to solve routing problems for logistics, supply chain management and similar industries. We design an algorithm to perform this table computation in the time-dependent scenario, where travel times depend on the departure time. There, the space-efficiency is very important, as an edge weight is described as a travel time function mapping the departure time to the travel time. They require much more space than a single real edge weight. Furthermore, the basic operations on edge weights are computationally more expensive on travel time functions than they are on single real values. Therefore, it pays off to perform additional computations to reduce the number of these operations. We will also show that we can develop interesting new algorithms for a variety of other problems requiring batched shortest paths computations. These problems have in common that a lot of distance computations are necessary, but not in the form of a table. The structure of the problem allows further engineering to improve the performance. For example, we develop a ride sharing algorithm that finds among thousands of offers the one with the minimal detour to satisfy a request within milliseconds. We improve the performance by limiting the maximum allowed detour. Another example is the fast computation of the closest points-of-interest.

Acknowledgements

This thesis would not have been written without the support of a number of people whom I want to express my gratitude. First in line is Peter Sanders, thank you for the numerous discussions and advice that helped me steering through my studies. Thank you, G. Veit Batz, for being a great office mate that was always available for interesting discussions, whether they were work-related or not. Thank you, Dominik Schultes, for introducing me into the world of route planning at the time I was still an undergraduate student. Thank you, Hannah Bast, for letting me be a part of your research team at Google, and for your willingness to review my thesis. Thank you, Erik Carlsson, Jonathan Dees, Daniel Delling, Arno Eigenwillig, Chris Harrelson, Moritz Kobitzsch, Dennis Luxen, Sabine Neubauer, Veselin Raychev, Michael Rice, Dennis Schieferdecker, Nathan Sturtevant, Vassilis Tsotras, Christian Vetter, Fabien Viger, and Lars Volker for great cooperation on various publications on diverse areas of route planning. Thank you, Fabian Blaicher, Sebastian Buchwald, and Fabian Geisberger for proofreading this thesis. Thank you, Anja Blancani and Norbert Berger for alleviating the burden of administration. My work has been partially supported by DFG grant SA 933/5-1 and the "Concept for the Future" of Karlsruhe Institute of Technology within the framework of the German Excellence Initiative.

Contents

1

Introduction

1.1 Motivation

Optimizing connections in transportation networks is a popular problem that arises in many different scenarios, such as car journeys, public transportation, or logistics optimization. People expect computers to assist them with these problems in a comfortable and fast way, whether they are at home, at work, or on a journey. This creates a demand for *efficient* algorithms to solve these problems. We see that in the existence of many online routing services, mobile navigation devices, and industrial tour planners.

The basic concept of a routing algorithm is to model the specific problem in a suitable *graph* and to compute a *shortest path* to solve it. While it is simple to come up with an algorithm that just solves the problem, it is much harder to engineer an efficient algorithm. Furthermore, the problems vary in different scenarios, making it impossible to create a single efficient algorithm for all of them. There are already efficient algorithms for basic scenarios, but the more advanced a scenario gets, the more attention is required to develop an efficient algorithm. For example, routing in public transportation networks is fundamentally different and much more difficult than routing in road networks, and therefore different algorithms are required. Of course, these algorithms also have some basic concepts in common, but much less than one would expect. Crucial for their efficiency is their specific development and engineering to a certain scenario.

The next two sections will cover problem statements and the current state of the art. We will begin with basic route planning, a largely solved problem, and then introduce advanced route planning problems and previously existing work.

1.2 Basic Route Planning

Basic route planning models the problem as a graph. The nodes of the graph represent geographic locations, such as junctions, and edges connect these locations, for example with roads. A valid connection in this model, from a source node to a target node, is a sequence of contiguous edges connecting source and target. Each edge is assigned a

non-negative weight, for example the length of the road or an estimation of the travel time required to reach from one end to the other. The optimization problem is to find a *shortest path* between a source node and a target node, that is a valid connection with *minimal* length (sum of edge weights).

In the last decade, most research focused on basic route planning in road networks, developing a plethora of increasingly faster *speed-up techniques*. Before that, only some classical algorithms existed that were not efficient on large graphs. The new faster algorithms usually perform a *precomputation step* for a graph that is independent of source and target nodes of subsequent queries. The auxiliary precomputation data helps to speed-up arbitrary shortest-path queries.

μήπως Κουκούτση ?

We judge the efficiency of these algorithms experimentally in the three-dimensional space of *precomputation time*, *precomputation space*, and *query time* compared on identical graphs. In each dimension, a smaller value is better than a larger one. A theoretical comparison of the fast algorithms is not possible, as there is no thorough theoretical work that matches the observed performance. A first attempt [3] to grasp the performance of some of the existing fast algorithms relies on a new graph property, that is, however, currently impossible to compute for large road networks, as it requires to solve NP-hard problems. Furthermore, this work does not allow to distinguish the performance of most of the examined fast algorithms. One reason is that constant factors are ignored in the comparison, but they are crucial for the practical usability of an algorithm on a specific graph.

The faster algorithms are divided into *hierarchical* approaches, *goal-directed* approaches, and *combinations* of both. In the following we will explain the most important algorithms shortly. Schultes [129] and Delling et al. [43] provide a chronological development of these algorithms, including some nowadays superseded ones.

Classical Algorithms

Dijkstra's Algorithm [50] is the fundamental shortest-path algorithm. It computes the shortest paths from a single source node to all other reachable nodes in the graph by maintaining *tentative distances* for each node. The algorithm *visits* (or *settles*) the nodes in order of their shortest-path distances. We can stop the search as soon as all target nodes are settled. No precomputation is required. Current implementations are based on a priority queue that maintains the tentative distances. In the worst-case, $O(n)$ values are extracted from the queue, and $O(m)$ values inserted or updated, where n is the number of nodes, and m is the number of edges in the graph. Therefore, an implementation using Fibonacci heaps [133] has a runtime of $O(m + n \log n)$ in the comparison based computation model. Sophisticated integer priority queues [135] can decrease the runtime. From a worst-case perspective, Dijkstra's algorithm largely solves the single-source shortest-path problem.

We usually compare faster algorithms by the speed-up over Dijkstra's algorithm, and if applicable, in the number of settled nodes. This provides experimental results that

can be compared to new algorithms mostly independent of the used hardware. A more detailed description of Dijkstra's algorithm is provided in Section 3.1.

A straightforward improvement of Dijkstra's algorithm is *bidirectional* search [35]. An additional search from the target node is performed in backward direction, and the whole search stops as soon as both directions meet. However, it can only be applied if the target node is known. Empirically, bidirectional search roughly halves the number of settled nodes.

A* Search [77] is a technique heavily used in artificial intelligence. It directs the search of Dijkstra's algorithm towards the target by using lower bounds on the distance to the target. Now, we always settle the node in order of their tentative distance from the source plus the lower bound to the target. The efficiency of this approach highly depends on the lower bounds. The simplest lower bound is based on the geographic coordinates of the nodes, but this results in poor performance on road networks. In case of travel time edge weights, even a slow-down of the query is possible [69].

Complete Distance Table. Precompute and store all shortest paths in the graph. This reduces a shortest-path query to a simple look-up, but requires massive precomputation time and space, and is therefore practically infeasible on large graphs.

Hierarchical Approaches

Reach-Based Routing. A node with high *reach* lies in the middle of long shortest paths [73]. The reach value is defined as the minimum of the lengths of the subpaths dividing the shortest path at the node, and maximized over all shortest paths. Then, a shortest-path search prunes nodes with a reach to small to get to source and target from there. The basic approach was considerably strengthened by an integration [70, 71] of *shortcuts* [123, 124], i. e., single edges that represent whole paths in the original graph.

Highway Hierarchies [123, 124, 129] creates a hierarchy of levels by alternating between node and edge contraction. Node contraction removes low-degree nodes and introduces shortcut edges to preserve shortest-path distances. The edge reduction then removes non-highway edges that only form a shortest paths of small length. The bidirectional query in this hierarchy subsequently proceeds only in higher levels. An improvement is to stop the contraction after only a small core graph is remaining, and to compute a complete distance table for the core.

Highway-Node Routing [130, 129] computes for a subset of nodes shortcut edges such that the subgraph induced by this subset of nodes preserves the shortest-path distances. This is done by performing a Dijkstra search from each node in the subset, until all computed paths to unsettled nodes contain a *covering node* in this subset. Shortcuts are

introduced to all covering nodes. We can recursively select a subset of nodes to obtain a hierarchy, with the smallest subset of nodes on top. A bidirectional query algorithm then never moves downwards in the hierarchy. The correctness is ensured by the shortcuts. To improve the efficiency of the algorithm, several pruning techniques have been developed.

Contraction Hierarchies [67, 59] intuitively assign a distinct "importance level" to each node. Then, the nodes are contracted in order of importance by removing them from the graph and adding shortcuts to preserve the shortest-path distances between the more important nodes. A bidirectional query only relaxes edges leading to more important nodes. Correctness is ensured by the added shortcuts. Therefore, it is similar to highway-node routing with n levels – one level for each node. However, the way shortcuts are added is different, and also the hierarchical classification. In comparison to highway hierarchies, contraction hierarchies are solely based on a more sophisticated node contraction. Interestingly, this algorithm is significantly more efficient than highway hierarchies. As node contraction is the most efficient hierarchical approach that can be adapted to many scenarios, we use it in our algorithms and give a more detailed description in Section 3.2.

Transit nodes are important nodes that cover long shortest paths. A complete distance table between all transit nodes is computed. For each node a subset of these transit nodes that cover long-distance shortest paths is computed, including their distance from the node. In road networks, this subset of nodes, called *access nodes*, is small. So a long-distance query can compute the shortest-past distance by just comparing all distances via each pair of access station of source and target node. The simpler short-distance queries are answered using another speed-up technique. This approach was first proposed by Bast, Funke and Matijevic [11, 12], with a simple geometric implementation. Shortly afterwards, Sanders and Schultes combined the approach with highway hierarchies [125, 13, 14]. Using contraction hierarchies to select transit nodes further improved the performance [67, 59].

Goal-Directed Approaches

ALT [69, 72] is based on **A***, **L**andmarks and the **T**riangle inequality. The A^* search is significantly improved on road networks when the lower bounds are computed using shortest-path distances from and to a small set of landmarks. Good landmarks are positioned behind the target, when viewed from the source, and vice versa. Therefore, they are selected at the far ends of the network. As A^* search just needs lower bounds, ALT can be easily adapted to many scenarios. Using ALT alone only gives mild speed-ups, but it is a powerful technique in combination with others. We use it in our algorithms and give a more detailed description in Section 3.3.

Edge Labels. The idea behind edge labels is to precompute information for an edge e that specifies a set of nodes $M(e)$ with the property that $M(e)$ is a superset of all nodes that lie on a shortest path starting with e. In a shortest-path query, an edge e needs not be relaxed if the target is not in $M(e)$. The first work specified $M(e)$ by an angular range [131]. Geometric containers [139, 141] provide better performance. Faster precomputation is possible by partitioning the graph into k regions with a small number of boundary nodes. Now $M(e)$ is represented as a k-vector of *edge flags*, also called *arc flags* [97, 96, 106, 107, 128, 81, 98, 80], where flag i indicates whether there is a shortest path containing e that leads to a node in region i.

Combined Approaches

A combination of the previously mentioned techniques usually results in a more efficient algorithm than a technique alone. A common classical scheme is to use a hierarchical technique and apply a goal-directed technique only on a *core* of the most important nodes identified by the hierarchical technique [131, 132, 83, 82]. This significantly reduces the preprocessing time and space of the goal-directed technique, and accelerates the query.

REAL [70, 71] is a combination of **RE**ach and **AL**t. Storing landmark distances only with the nodes with high reach values can reduce memory consumption significantly.

SHARC [20, 21, 38] combines **SH**ortcuts and multi-level **ARC** flags. Shortcuts allow to unset arc flags of edges that are represented by the shortcut, reducing the search space. The query algorithm can be *unidirectional*, which is advantageous in scenarios where bidirectional search is prohibitive. But a bidirectional query algorithm is faster in the basic scenario.

Core-ALT [22, 127, 23] iteratively contracts nodes that do not require too many shortcuts. Then, on the remaining **C**ore, a bidirectional **ALT** algorithm is applied. As source and target node of a query are not necessarily in the core, *proxy nodes* in the core [71] are used. The best proxy nodes are the core entry and exit node of a shortest path for source to target. However, as the query wants to compute a shortest path and does not know it in advance, the core nodes that are closest to source and target are used.

CHASE [22, 127, 23] combines **C**ontraction **H**ierarchies and **A**rc flag**S**. First, a complete contraction hierarchy is created. Then, on a core of the most important nodes, including shortcuts, arc flags are computed. The query needs only use the edges with arc flags set for one of the regions where the other search direction has an entry node into the core. This results in a very fast algorithm, only algorithms based on transit nodes are faster.

Transit Nodes + Arc Flags. The transit nodes approach is made even faster by computing flags for each access station [22, 23]. In a sense, this computes arc flags when the set of access station is viewed as a set of shortcut arcs to them. The speed-up over Dijkstra's algorithm is more than 3 000 000 on a Western European road network.

1.3 Advanced Route Planning

Basic routing algorithms struggle with advanced routing scenarios, so there is a need to algorithmically enhance them or even to develop completely new algorithms. The need for such advanced scenarios stems from diverse advantages:

- Basic routing only considers a simple edge weight function that insufficiently models reality. For example, travel times change depending on the traffic conditions. With historical traffic data and traffic prediction models, it is possible to forecast travel times more accurately.
- The limitation to a single edge weight function is not desirable for a user that likes to have choices. Different edge weights allow to optimize for different criteria, for example travel time, fuel economy, or distance. Furthermore, trade-offs in such criteria are important, as usually more than one criterion is important.
- Not all routes are open to every type of vehicle, or a fee needs to be paid. So the computation of a shortest path with individual restrictions on certain routes is a desirable feature.
- Routing in public transportation networks is inherently time-dependent, and requires multi-criteria optimization, as at least the travel time and the number of transfers are important. Quickly finding good routes makes using public transit more comfortable and helps to increase their popularity.
- Certain problems, such as the vehicle routing problem, require the computation of a large number of related shortest paths. This relation can be exploited to speed-up their computation.

Further advanced scenarios, that are not considered in this thesis, are:

- Dynamic routing considers current traffic situations, such as traffic jams [130, 129].
- Alternative route computation provides flexible and reliably routes, and choices to the user [2, 36].
- Multi-modal routing considers switching between types of transport, for example cars and planes [42, 119].

- Traffic simulations require massive numbers of shortest-path computations that cannot be handled by a single computer. Parallel and distributed algorithms become necessary [89, 90].

- In mobile scenarios, the routing algorithms have to run on very limited hardware. Especially main memory is small, and therefore external memory needs to be considered [72, 126, 137].

- Turn restrictions and turn costs increase the detail of the model. They can usually be incorporated into existing algorithms by an edge-based graph [31, 142, 138].

1.3.1 Time-dependency in Road Networks

Time-dependent edge weights model time-dependency in road networks [33, 54, 91]. Such a weight is a travel time function (TTF) that maps the departure time at the source node of the edge to the travel time required to reach the target node of the edge. We distinguish between two types of queries. A *time* query computes the earliest arrival time at a target node, given the departure time at the source node. This problem is also known as the earliest arrival problem (EAP). A *profile* query computes a *travel time profile* for all departure times.

Dijkstra's Algorithm is easily extended to perform a time query in case that the *FIFO-property* is fulfilled: it is not possible to arrive earlier when departing later [33]. We always use the reasonable assumption that our road networks fulfill the FIFO property. If not, and waiting is allowed, the problem is NP-hard [116]. To compute a whole travel time profile, Dijkstra's algorithm can be extended to iteratively correct tentative travel time functions [116]. However, this algorithm loses its node settling property.

A few basic speed-up techniques have been augmented to cope with time-dependent edge weights. The general scheme is to ensure that a property is valid for all departure times. Furthermore, bidirectional time queries are more difficult, as the arrival time is not known in advance. Also, as the travel time functions are much more complex than time-independent travel times, additional important algorithmic ingredients significantly improve the efficiency of the algorithms.

ALT can be adapted by performing the landmark computation on static lower bounds of the travel time functions [41]. Therefore, the A^* potentials provide valid lower bounds for all departure times. Still, the performance is not very good.

Core-ALT performs the node contraction so that the earliest arrival times between the remaining nodes are preserved for all departure times [41, 38]. The bidirectional query is more complex, as the arrival time at the target node is not known. Therefore, the

backward search uses lower bounds to reach the core, and the search in the core is only performed in forward direction.

SHARC is generalized by using time-dependent arc-flags [37, 39, 38]. An arc flag of an edge is set if there is a single departure time where the edge is on a shortest path to the target region. Its unidirectional query has advantages over bidirectional algorithms, as the *arrival time* is not known. A space-efficient variant [30] further reduces the memory overhead, especially by not storing the travel time functions of shortcuts, but computing them on demand. Very fast is an inexact variant, that computes the arc flags only for a few departure times. However, no approximation guarantee for the query result is known.

Contraction Hierarchies use an improved node contraction routine and consider the complexity of the travel time functions in the node order computation [17, 15, 136]. Using approximate travel time functions significantly decreases the space overhead while still producing exact results [113, 16]. The trick is to use lower and upper bounds that allow to compute *corridors* including the shortest path. The most efficient profile queries are performed by contracting these corridors. Queries become faster but inexact, when only based on approximate travel time functions. However, we are the first to prove approximation guarantees for these queries in Section 6.2.4.

1.3.2 Public Transportation

Speed-up techniques are very successful when it comes to routing in time-dependent road networks [46]. However, there is only little previous work on speed-up techniques for public transportation networks, and none of it as successful as for road networks. Timetable networks are very different from road networks, and the techniques used for road networks usually do not work for public transportation networks [9]. There is limited success in transferring these techniques in simple scenarios on well-structured graphs, but there is no success for fully realistic scenarios on graphs with poor structure.

Scenarios. Routing in public transportation networks can be divided in further scenarios. A scenario describes, which *features* the query algorithm supports. The following features seem natural for a human that uses public transportation, but each of them has influence on the routing algorithm and the success of the speed-up technique. We consider the following features:

- *minimum transfer duration:* If a transfer happens at a station, there has to be a minimum transfer duration between the arrival of the first train, and the departure of the second train.

- *walking between stations:* A transfer cannot only happen at a single station, but it is possible to walk to a nearby station to board the next train.
- *traffic days:* There can be a different timetable every day of our planning horizon. We try to store this information efficiently instead of adding a separate connection for each day. If this feature is not supported, the same timetable is used every day.
- *multiple criteria:* This feature considers several independent optimization criteria. In addition to the earliest arrival time, we also optimize the number of transfers, walking distance or the like. This can result in more than one optimal result.
- *location-to-location queries:* Location-to-location queries consider an arbitrary start and end location for a query, instead of a start and end station. It includes the walking to potentially several nearby start stations and the walking from several nearby end stations to the end location.

The scenario with *realistic transfer durations* supports only minimum transfer durations. This is the simplest scenario where the computed connections can be used in reality, but still may be not optimal from a human perspective. Nevertheless, speed-up techniques for road networks already start to fail just by considering minimum transfer durations.

The *fully realistic scenario* supports all of the above features. It is the desired scenario for a public transportation router used in practice. However, all the supported features make it very hard, and especially for poorly structured networks, completely new query algorithms are necessary.

Models. A *model* describes how to create a graph from the timetables such that we can answer queries in this graph by shortest-path computations. It is possible to support each scenario feature in each of the described models, but with different effects on the efficiency of speed-up techniques. Furthermore, there are usually a lot of variants of these models, so we will restrict us to the most relevant variant of each model. More details on models provides Section 2.3.

In the *time-expanded model* [110, 102, 132], each node corresponds to a specific time event (departure or arrival), and each edge has a constant travel time. To allow transfers with waiting, additional transfer nodes are added. By adding transfer edges from arrival nodes only to transfer edges after a minimum transfer duration passed, realistic transfers are ensured. The advantage of this model is its simplicity, as all edges weights are simple values, and Dijkstra's algorithm can be used to compute shortest paths.

The *time-dependent*[1] *model* [29, 111, 116, 117] reduces the number of nodes in comparison to the time-expanded model, that showed to be a performance obstacle. The stations are expanded to a *train-route graph* [121]. A *train route* is a subset of trains

[1]Note that the time-dependent model is a special technique to model the time-dependent information rather than an umbrella term for all these models.

that follow the exact same route, at possibly different times and do not overtake each other. Each train route has its own node at each station. Those are interconnected within a station with the given transfer durations. As there are usually significantly fewer train routes at a station than events, this model reduces the number of nodes. However, this model also becomes more complex, as the edges between the train route nodes along a train route are time-dependent.

A *station graph model* uses exactly one node per station, even with support for minimum transfer durations. Berger et al. [26, 25] introduced such a model by condensing the time-dependent model. This results in parallel edges, one for each train route, and their query algorithm computes connections per incoming edge instead per node. Their improvement over the time-dependent model is mainly that they compare all connections at a station and remove dominated ones. We independently developed a different station graph model, and will emphasize the differences to the model by Berger et al. [26, 25] upon its introduction in Section 4.1.

Previous Speed-up Techniques. Goal-directed search (A*) brings basic speed-up [77, 120, 121, 52] and can be adapted to all scenarios and models.

Time-dependent SHARC [39, 38] brings better speed-up by using arc flags in the scenario with realistic transfer durations. It achieves query times of a few milliseconds but with preprocessing of several hours.

The fully realistic scenario was recently considered by Disser et al. [52] and Berger et al. [25]. However, the network considered in those papers is relatively small (about 8900 stations) and very well-structured (German trains, almost no local transport). Also, there are only very few walking edges, as walking between stations is rarely an issue for pure train networks. Disser et al. [52] reported query times of about one second and Berger et al. [25] of a few hundred milliseconds. The title of the latter paper aptly states that obtaining speed-ups for routing on public transportation networks in a realistic model "is harder than expected".

1.3.3 Flexible Queries in Road Networks

In the *flexible* scenario the graph is enriched with additional attributes compared to basic route planning. These attributes are taken into account by the shortest-path computation by using additional query *parameters*. We consider *multiple edge weights* and *edge restrictions*. Formal details are provided in Section 2.2.3.

Multiple Edge Weights

With multiple edge weights there is usually no single optimal path, as the paths have different *costs* in the different weights. In the classic approach all *Pareto-optimal* paths P are computed, i. e., where for each other path P', there is at least one of the edge weights for which P is better than P'. A path P is said to *dominate* another path P' if

P' is not better in any edge weight. The most common algorithm to compute all Pareto-optimal paths is a generalization of Dijkstra's algorithm (*Pareto-Dijkstra*) [76, 103]. It does no longer have the node settling property, as already settled nodes may have to be updated again, and multiple Pareto-optimal paths to a node are represented by multi-dimensional *labels*. Computing all Pareto-optimal paths is in general NP-hard, and in practice exist also a plethora of Pareto-optimal paths on road networks.

Even though the Pareto-optimal shortest path problem has attracted more interest in the past, the parametric shortest path problem [87] has also been studied. However, it provides less flexibility as compared to the approach we will introduce in Section 2.2.3. Given a value p, we are just allowed to subtract p from the single weight of a predefined subset of edges. All well-defined shortest path trees (when p is too large, we may get negative cycles) can be computed in $O(nm + n^2 \log n)$ [118].

Previous Speed-up Techniques. To the best of our knowledge, the most successful result on speed-up techniques for multi-criteria is an adaptation [45, 38] of the SHARC algorithm [20, 38]. However, Pareto-SHARC only works when the number of optimal paths between a pair of nodes (*target labels*) is small. Pareto-SHARC achieves this either by very similar edge weight functions or by tightening the dominance relation that causes it to omit some Pareto-optimal paths (*label reduction*). Yet, the label reduction entails serious problems as not all subpaths of a path that fulfills those tightened domination criteria have to fulfill the criteria themselves. Therefore, their algorithm may rule out possibly interesting paths too early, as it is based on a Dijkstra-like approach to compute optimal paths from optimal subpaths. Thus, Pareto-SHARC with label reduction cannot guarantee to find all optimal paths w. r. t. the tightened domination criteria and is therefore a heuristic. In a setup similar to the one we use in Section 5.2, they can only handle small networks of the size of a city. As simple label reduction, they propose to dominate a label if the travel time is more than ε times longer the fastest. This is reasonable but only works well for very small $\varepsilon \leq 0.02$ with around 5 target labels. Even for slightly larger ε, the preprocessing and query times increase significantly. Also, stronger label reduction is applied to drop the average number of target labels to 5 even for $\varepsilon = 0.5$. It seems like Pareto-optimality is not yet an efficient way to add flexibility to fast exact shortest path algorithms.

Edge Restrictions

In this scenario, edges have additional attributes that restrict their usage in a query. For example, a user wants to find the shortest path ignoring all unpaved roads. Therefore, a query algorithm needs to find a shortest path while ignoring a query-specific set of edges. Dijkstra's algorithm is able to do that with straightforward modifications.

Previous Speed-up Techniques. A naïve adaption of a basic speed-up technique just repeats the precomputation separately for each choice of restrictions. However, this is

usually infeasible for a large number of different restrictions, as there is an exponential number of possibilities to combine them. Only ALT [69, 72] and other A^*-based algorithms can be directly used, as removing edges does not invalidate their lower bounds on the shortest path distances [44].

1.3.4 Batched Shortest Paths Computation

In the *batched* scenario, shortest path distances are computed for multiple source-target pairs. However, there are usually much less different source and/or target nodes than there are source-target pairs. This can be exploited to design special algorithms that speed-up the computations.

Dijkstra's algorithm is especially strong in the batched scenario, as it starts from a single source node and can compute the distances to multiple target nodes. We then stop the search only after all target nodes are settled. If the shortest-path distances to all nodes in the graph are required, Dijkstra's algorithm is even among the most efficient ones. A very inefficient part in its execution is settling a node whose shortest-path distance is not required.

Previous Speed-up Techniques. Knopp et al. [93, 92, 94, 129] where the first to accelerate the computation of distance tables beyond Dijkstra's algorithm. They observed that a *bidirected* and *non-goaldirected* speed-up technique repeatedly performs the same forward or backward search when the source or target node does not change. So they first compute the backward search from each target node, storing the tentative distances at the settled nodes together with the information about the target node. Then, a forward search from a node can derive the shortest-path distances to all target nodes by using the stored distances at each settled node like in a regular bidirectional query. Currently the fastest suitable speed-up technique is contraction hierarchies [67, 59]. We generalize this idea in Chapter 6.

Applications

We tailored efficient algorithms for important real-life problems that we will introduce here together with related work.

Time-dependent Vehicle Routing. We provide an algorithm that augments the computation of large travel time tables to the time-dependent scenario. These tables are important for the optimization of fleet schedules. It is known as the vehicle routing problem (VRP), which is an intensively studied problem in operations research [27, 28], but also experiences growing attention from the algorithm community [99, 88]. As the VPR is a generalization of the traveling salesman problem, it is NP-hard. The goal is to find routes for a fleet of vehicles such that all customers (locations) are satisfied and the total

cost is minimized. Solving this problem is important for logistics, supply chain management and similar industries. In the time-dependent scenario, this problem is known as the time-dependent vehicle routing problem (TDVRP), and there exist many algorithms to solve it [101, 85, 84, 53, 79]. The goal is the same as for VRP, but now, the costs are time-dependent. Industrial applications often compute travel times for a discrete set of departure times, e. g. every hour. This approach is very problematic as it is expensive to compute, requires a lot of space (a table for every hour), and provides absolutely no approximation guarantee. Our approximate variants do not have these disadvantages. We require less precomputation time and space, an important aspect for companies because they can run the algorithm on smaller and cheaper machines. And, even more important, we provide approximation guarantees which potentially results in better routes in practice that further reduce the operational costs of their transportation business.

Ride Sharing. Ride sharing is a concept where a driver (with a car) and a passenger (without a car) team up so that the driver brings the passenger to her ending location. As they usually do not share the exact same starting and ending location, the detour of the driver should be small. The passenger will pay some money to the driver, but less than the cost of traveling on her own, and therefore both save money. Also, environmental aspects play an important reason for sharing a ride.

There exists a number of web sites that offer ride sharing matching services to their customers. But they all suffer some limitations. The most common approach is to only allow starting and ending locations to be described imprecise from a predefined set, for example by simply the city name or some points of interest like airports. Some of the web sites improve this by offering radial search around starting and ending locations to increase the number of matches. Still, these approaches ignore a lot of reasonable matches.

Our idea is to match driver and passenger by the detour for the driver to pickup the passenger at her starting location and bring her to her ending location. We rank the matches by detour, preferring smaller ones. The detour can be small, even when both starting and ending locations are not close. To the best of our knowledge, there exists no previous work on fast detour computation, which would enable drivers and passengers to meet somewhere in between.

To compute the detours, it is necessary to know for each offered drive the distance to the passengers pickup location and the distance to the ending location after dropping the passenger. For the first set of distances, the target node is always the starting location of the passenger, for the latter set, the source node is always the ending location of the passenger. Dijkstra's algorithm, a basic speed-up technique (Section 1.2), or a fast table computation algorithm can be used. But we will present a more efficient algorithm.

There exists also research on ride sharing outside the algorithmic community. Several authors [51, 115, 78] investigated the socio-economic prerequisites of wide-spread customer adoption and overall economic potential of ride sharing. Other authors [144, 143] propose to use hand-held mobile devices to find close-by drivers/passengers.

Xing et al. [144] gave an approach to ad-hoc ride sharing in a metropolitan area that is based on a multi-agent model. But in its current form the concept does not scale. As the authors point out it is only usable by a few hundred participants and not by several thousands or more participants that a real world ride sharing service would have.

Point-of-Interest Location. There are applications that require to compute close-by points of interest (POI), for example a mobile navigation device should compute all close-by Italian restaurants. The simplest approach to compute them is based on Dijkstra's algorithm, as it settled nodes in ascending distance from the source. However, this is only feasible for very close POI, as any settled node that is not a POI is wasted effort. Also, the computation of POI close to a path becomes more expensive with Dijkstra's algorithm, especially when this path is long. Therefore, spatial data structures, such as quadtrees or R-trees [74] are used to find geographically close POI. The downside of this approach is poor performance as there is only low correlation between geodistance and shortest-path distance, especially when travel time metric is used. To filter or order the geographically close POI by true shortest-path distance, separate shortest-path queries are necessary. This makes this approach computationally expensive and we provide a more efficient algorithm that does not rely on spatial data structures.

1.4 Main Contributions

1.4.1 Overview

We present new efficient algorithms in three main areas of advanced route planning. These algorithms are significantly more efficient than any other previous algorithm, often they even solve a problem efficiently and exact for the first time. Moreover, we do not only create new algorithms, but also new models and problem refinements that allow better algorithms without losing track of the purpose behind the initial problem definition.

1.4.2 Public Transportation

As stated in Section 1.3.2, in recent years, the advancement of algorithms for public transportation progressed much slower than for road networks. The techniques that work excellently on road networks largely fail on public transportation networks. Therefore, it is both interesting to get a better understanding why these techniques fail, and to develop completely new ones tailored to public transportation networks.

Routing with Realistic Transfer Durations. This is the simplest scenario where the techniques for road networks start to fail. We focused on the hierarchical method of node contraction and discovered, that although networks can be contracted, the standard

models to create the graph from a timetable cause problems. They use multiple nodes for a single station, allowing to have "parallel" edges between different nodes of the same station. This significantly hinders the performance of node contraction. The reason for multiple nodes per station is that the edge weights can then be simpler. We introduce a new *station graph model* with just a single node per station, no parallel edges, and more complex edge weights. Because of the minimum transfer durations, not only the earliest arriving train at a station is the best one, but to continue a journey, other trains that arrive later but require no transfer may be better. This made the development of efficient algorithms for the required operations on these complex edge weights the most difficult part of our research. However, further augmentations of the node contractions, such as some sort of caching, additionally improve the overall performance.

Fully Realistic Routing. In practice, routing in transportation networks is a multi-criteria and multi-modal problem. Next to an early arrival time, at least a small number of transfers, and short walking distances at transfers are important. This increases the complexity of the problem to a point where all previous techniques for routing in road networks fail to significantly speed-up queries. Therefore, Hanna Bast, at this time a visiting scientist at Google Zürich, and her team came up with the idea of *transfer patterns*: store for a pair of source and target station only the transfer patterns of all optimal connections, that is the sequence stations where transfers happen. Even with thousands of optimal connections between a pair of stations, we can expect usually a small set of transfer patterns describing them. Based on all optimal transfer patterns between a pair of stations, a *query graph* is created. Each edge represents a *direct connection* without any transfers in between. Answering such direct-connection queries is much simpler, and allows to efficiently perform a search on the query graph. The author of this thesis joined the team with the task to develop an efficient algorithm to precompute the transfer patterns. This, at first glance, simple task turned out to be very challenging. A naïve algorithm would perform a single-source search from each station to compute the transfer patterns for all departure times. However, this is infeasible on large networks with hundreds of thousands of stations, as they are used at Google.

Therefore, the idea of *hubs* came up that should cover the searches from the other stations. This idea stems from the transit nodes approach of road networks and works well there. In more detail, each non-hub only knows the transfer patterns to all relevant nearby hubs. The search starting at such a non-hub can be stopped as soon as all further computed paths include a transfer at a hub. To obtain optimal transfer patterns to the target station, we append the transfer patterns computed from the hub to the target station. But the structure of public transportation networks is much more difficult than the one of road networks. There is almost always a path without a transfer at a hub that takes hours, hindering the local search to stop early. Effectively, the hubs approach reduces the precomputation time by less than 10%. So we had to take the hard decision to drop exactness and to rely on heuristics. The most important one is the three-legs heuristic that prunes a local search before the third transfer happens. Any connection with more

than two transfers to reach a hub anyway looks suspicious to a human. As expected, this creates not more errors than already present due to modelling errors. Further improvements reduce the number of labels computed by the local and global searches, and reduce the number of the computed transfer patterns without compromising the quality of the computed paths. The latter one is especially important for fast queries, as it reduces the size of the query graph.

The final algorithm is in use for public transportation routing on Google Maps (`http://www.google.com/transit`). We extensively tested it on different public transportation networks, some of them being very large and having poor structure.

1.4.3 Flexible Queries in Road Networks

Flexible queries provide diverse problems that mostly have not been tackled before from the viewpoint of an algorithm engineer. Still, they deserve our attention, as they are highly relevant in practice and a major reason why the current algorithms have not been adapted widely in industry.

Multiple Edge Weights. Previous research on fast routing in road networks focused mainly on a single metric. Mostly the travel time metric was used, sometimes also other metrics such as travel distance. Almost all work on multiple metrics focuses on Pareto-optimality, but this provides no truly fast algorithms, as computing several paths at once is significant overhead, and much slower than just a multiple of single-criterion queries. Therefore, we came up with the idea to linearly combine two metrics with a single parameter representing the coefficient. Think of this parameter as a slider, that allows to select the trade-off between the two metrics under consideration for each query. This provides significantly more flexibility than a single-criterion scenario, but allows an efficient precomputation. We do not need to perform the precomputation for each parameter value separately by exploiting certain properties of the linear combination.

As these metrics result in different shortest paths, the hierarchical classification of the network differs between them. Therefore, we develop the concept of *parameter interval splitting*: We use different classifications for the most important nodes, depending on an interval of parameter values, instead of using a single hierarchy for the whole interval of parameter values. The less important nodes are treated the same, thus saving precomputation time and space.

To improve the speed of precomputation and query, we combine our hierarchical approach with goal-direction based on landmarks. The problem there is to compute feasible potentials for any choice of parameter value. Again, properties of the linear combination of edge weights can be exploited. We can compute feasible potentials for a whole interval of parameter values by just precomputing potentials for the border values of the interval.

To be able to compute all shortest paths within an interval of parameter values, we developed efficient algorithms that require a number of flexible queries that is linear in the number of different paths, and otherwise independent of the size of the interval.

Edge Restrictions. Not all streets are the same, for example highways allow faster travel, toll roads cost money, and the transport of hazardous goods is forbidden on roads in water protection areas. While different speed limits can be incorporated into the travel time metric, other restrictions cannot. We want an algorithm that allows on a per query basis to select the restricted edges by their properties.

The first adaption of node contraction to this scenario was done by Michael Rice and Vassilis Tsotras at the University of California in Riverside. We decided to join forces to increase the performance of their initial algorithm and to add new features. Under the lead of the author of this thesis, significant improvements happened: We accelerate the precomputation by augmenting the node contraction to its full potential as in the basic scenario. By adding support for parametrized restrictions, such as the maximum allowed height of a vehicle, the algorithm becomes even more flexible. Combinations with goal-directed techniques improve the precomputation time, and the query time. By computing landmarks for different sets of restrictions, the potentials are improved for more restricted queries.

An interesting observation is that more restricted queries can be answered faster. The reason is that they relax fewer edges, as they restrict a lot of them. However, a less restricted query relaxes certain shortcut edges in vain. Those shortcuts are only necessary to preserve the shortest-path distances for more restricted queries. We therefore introduce the concept of *witness restrictions* that reduces the number of unnecessarily relaxed shortcuts.

1.4.4 Batched Shortest Paths Computation

Computing multiple related shortest-path distances at once is necessary for many real-life problems. The most popular one is the computation of a full distance table for vehicle routing optimization. Exploiting the relation between the source and target nodes of the distances leads to significantly faster algorithms. But we show that also other related interesting problems benefit from special algorithms.

Time-dependent Travel Time Table. We augment the algorithmic ideas of the static table computation algorithm to the time-dependent scenario. While some of the old ideas can be kept, we need to develop several additional ideas to cope with the more complex scenario. Time-dependency is modeled by travel time functions as edge weights. As they require much more space than a simple edge weight that is usually represented by a single integer, we chose to refine the problem of computing a table to the implementation of a query interface. By that, we can provide an algorithm that requires precomputation

time and space linear in the number of source and target nodes of the table, while still being more than one order of magnitude faster than competing algorithms.

Previously simple operations on edge weights (add, min on integers) map to expensive operations on travel time functions in the time-dependent scenario. Therefore, we develop exact techniques to replace most of the expensive operations by cheaper ones. This also requires changes to the data organization of the static table computation algorithm. Furthermore, we provide several algorithms that store different amounts of data to accelerate queries. These algorithms provide different trade-offs between preprocessing time, preprocessing space, and query time.

Approximate versions of these algorithms significantly improve the performance and the space requirements of our algorithms. This is especially important for industrial applications. However, the simple heuristics already used there do not provide any approximation guarantee. To the best of our knowledge, our algorithms compute tables faster, require less space, and allow to select worst-case approximation bounds. We tested our algorithms on different road networks, and for low and high traffic scenarios.

Ride Sharing. Ride sharing problems have not been considered before from an algorithmic point of view. Current systems are largely based on the features of a database, limiting new approaches. There, matching a driver having a car, and a passenger to share a ride is based on the proximity of their starting and ending location. However, this concept limits the number of matches, as it is not possible that the driver picks up the passenger on her trip, even when only a small detour is required. We present a new algorithm for efficient detour computation, that compares hundreds of thousands of offers in a database within milliseconds. We further prune the computation by limiting the maximum detour we are interested in, a very reasonable measure in practice. This pruning is not straightforward and requires deep knowledge of the algorithmic details. Our new algorithm allows significantly more matches over the previous approaches, hopefully leading to a wider popularity of ride sharing in the population.

Point-of-Interest Location. To compute points of interest (POI) closest to a point, we need to compute the shortest-path distances from this point to all POI. While Dijkstra's algorithm seems just perfect for this problem, we exploit hierarchical speed-up techniques to even further speed-up the computation. We do that by precomputing the search spaces from all POI. A query then only needs to compute the single search space from the point and finds all interesting POI. By limiting the distance to the POI, we can further prune the computation. Even more interestingly, we can efficiently compute POI closest to a shortest path by adapting the detour algorithm developed for ride sharing. For example, this allows to compute the gas station with the smallest detour to the initial target. To the best of our knowledge, current navigation systems do not consider the target for POI location. Our algorithm therefore allows interesting new approaches for such systems.

1.5 Outline

The remaining chapters of this thesis are organized as follows:

Chapter 2 introduces basic definitions and fundamentals of graph theory. In more detail, different scenarios and models for road and public transportation networks are recaptured from literature.

Chapter 3 introduces some basic concepts that frequently occur in our algorithms.

Chapter 4 is devoted to routing in public transportation networks. We distinguish between two major scenarios: routing with realistic transfer durations, and fully realistic routing. For both scenarios, we present new efficient algorithms and test them on different input networks.

Chapter 5 presents efficient algorithms for two different flexible query scenarios. First, a flexible algorithm that supports two objective functions. Second, we consider edge restrictions. Both algorithms are extensively experimentally evaluated.

Chapter 6 considers several application oriented problems that require multiple combined shortest-path computations in the areas of time-dependent travel time tables, ride sharing, and point of interest location. We present several algorithms for the travel time table computation and perform an extensive experimental comparison. Our approach of detour minimization for ride sharing is introduced, and an efficient algorithm presented. We experimentally analyze both our new approach and the performance of our algorithm. Finally, we contribute new algorithms to compute points of interest close to a point or a path, and evaluate their efficiency.

Chapter 7 concludes our work, which also contains some notes on possible future work and an outlook.

2

Fundamentals

In this chapter, we provide the fundamental concepts that we base our work on. These are all related to graphs, but extend the basic understanding of a graph in different directions.

2.1 Graphs and Paths

A (directed) *graph* $G = (V, E)$ is defined by a *node* set V of size n and an *edge* set E of size m. Every edge connects two nodes, the *source* and the *target*. Furthermore, depending on the scenario, an edge may have several additionally attributes. Usually, an edge $e \in E$ is uniquely identified by its source $u \in V$ and target $v \in V$, and we just write (u, v) instead of e. In case that there are *parallel edges* that have same source and target (see Chapter 5), additional attributes of an edge, or the current context are necessary for a unique identification.

A path P in G is a sequence of edges $\langle e_1, \ldots, e_k \rangle$ such that the target of e_i and the source of e_{i+1} are the same for all $1 \leq i \leq (k-1)$. In case that our graph has no parallel edges, or if it is clear from the context, we can represent a path also by the node sequence $\langle u_1, \ldots, u_k, u_{k+1} \rangle$, where u_i is the source of edge e_i, for all $1 \leq i \leq k$, and u_{k+1} is the target of e_k. We call u_1 the *source of* P and u_{k+1} the *target of* P.

2.2 Road Networks

In road networks, nodes usually represent junctions, and edges represent road segments.

2.2.1 Static Scenario

In the static scenario, each edge is weighted by a function $c : E \to \mathbb{R}_+$, and no parallel edges exist. For an edge $e = (u, v)$ we also write $c(u, v)$ instead of $c(e)$. We call such a simple weighted graph *static*. The edge weight usually represents the average travel time required for the road segment, or its physical length. The length of a path P is $c(P) = \sum_{i=1}^{k} c(e_i)$. A path P^* is a *shortest path* if there is no path P' with same source and

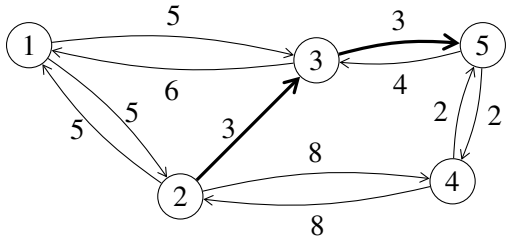


Figure 2.1: Static graph with $n = 5$ nodes and $m = 11$ edges. The edges are annotated with their weight. The shortest path from source node 2 to target node 5 is thick.

target as P^* such that $c(P') < c(P^*)$. The *(shortest-path) distance* $\mu(s,t)$ is the length of a shortest path with source s and target t, or ∞ if there is no such path. Figure 2.1 shows a static graph.

2.2.2 Time-dependent Scenario

To model more realistic travel times, our edge weights in this scenario depend on the departure time. More formally, the edge weight $c(u,v)$ of an edge $(u,v) \in E$ is a function $f : \mathbb{R} \to \mathbb{R}_{\geq 0}$. This function f specifies the time $f(\tau)$ needed to reach v from u via edge (u,v) when starting at *departure time* τ. So the edge weights are called *travel time functions* (TTFs). For convenience, we will write $c(u,v,\tau)$ for $c(u,v)(\tau)$.

In road networks we usually do not arrive earlier when we start later. So all TTFs f fulfill the *FIFO-property* [33]: $\forall \tau' > \tau : \tau' + f(\tau') \geq \tau + f(\tau)$. In this work all TTFs are sequences of *points* representing piecewise linear functions. This representation is the simplest model supporting the FIFO-property, and that is closed concerning the necessary operations stated below. Usually, a piecewise linear function is continuous. However, if we represent events such as the departure of a train or a ferry, then points of discontinuity exists exactly at these departure times. Note that piecewise constant functions do not support the FIFO-property.

We assume that all TTFs have period $\Pi = 24\text{h}$. However, using non-periodic TTFs makes no real difference. Of course, covering more than 24h will increase the memory usage. This enables us to represent the functions as finite sequences of points $\langle (x_1,y_1),\ldots,(x_k,y_k)\rangle$ with $0 \leq x_1 < \cdots < x_k < \Pi$. With $|f|$ we denote the *complexity* (i. e., the number of points) of f.

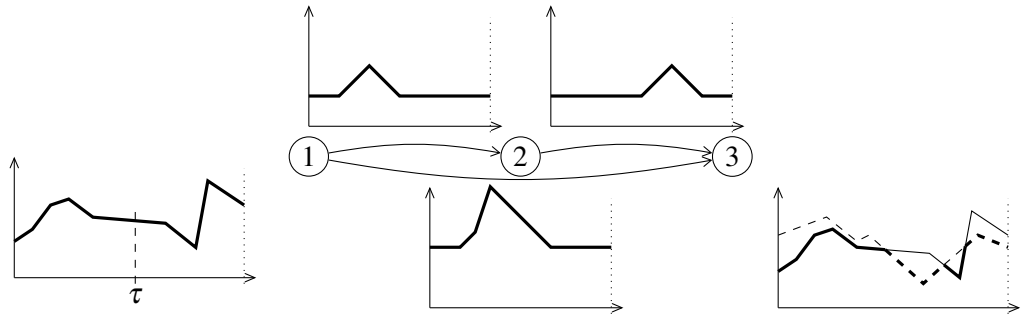
But any representation of TTFs is possible that supports the following three operations (Figure 2.2):

- *Evaluation.* Given a TTF f and τ we want to compute $f(\tau)$. Using a bucket structure this runs in constant average time.
- *Linking of TTFs.* Given a path $P = \langle u,\ldots,v\rangle$ with TTF $f := c(P)$ and a path $Q = \langle v,\ldots,w\rangle$ with TTF $g := c(Q)$, we want to compute the TTF of the path

$\langle u,\ldots,v,\ldots,w\rangle$. This is the function $g*f:\tau\mapsto g(f(\tau)+\tau)+f(\tau)$.[1] It can be computed in $O(|f|+|g|)$ time and $|g*f|\in O(|f|+|g|)$ holds. On the one hand linking is an associative operation, i. e., $f*(g*h)=(f*g)*h$ for TTFs f,g,h. On the other hand linking is *not* commutative, i. e., $f*g\neq g*f$ in general.

- *Minima of TTFs.* TTFs f,f' from u to v, we want to *merge* these into one while preserving all shortest paths. The resulting TTF from u to v gets the TTF $\min(f,f'):\tau\mapsto\min\{f(\tau),f'(\tau)\}$. It can be computed in $O(|f|+|f'|)$ time and $|\min(f,f')|\in O(|f|+|f'|)$ holds. The minima operation is associative and commutative, as $\min(f,\min(g,h))=\min(\min(f,g),h)$ and $\min(f,g)=\min(g,f)$ holds for TTFs f,g,h.

The link and minima operation are distributive, i. e., for TTFs f, f' and g holds $\min(g*f,g*f')=g*\min(f,f')$ and $\min(f*g,f'*g)=\min(f,f')*g$.



(a) evaluation at τ (b) linking $1\to2\to3$ (c) minimum (thick)

Figure 2.2: Operations on travel time functions.

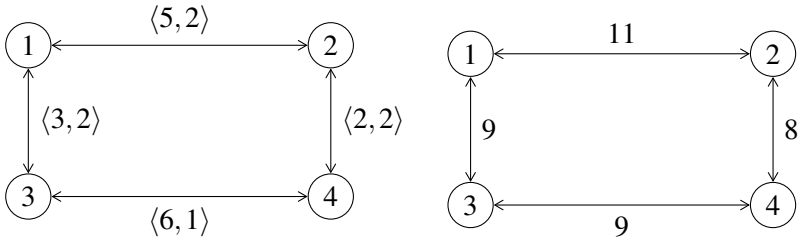
In a time-dependent road network, shortest paths depend on the departure time. For given start node s and destination node t there might be different shortest paths for different departure times. The shortest-path length from source node s to target node t with departure time τ is denoted by $\mu(s,t,\tau)$. The minimal travel times from s to t for all departure times τ are called the *travel time profile* from s to t and are represented by a TTF denoted by $\mu(s,t)$.

We define $f\sim g:\Leftrightarrow\forall\tau:f(\tau)\sim g(\tau)$ for $\sim\in\{<,>,\leq,\geq\}$.

2.2.3 Flexible Scenario

A *flexible graph* is a graph $G=(V,E)$ with additional attributes. Furthermore, a shortest path is not only defined by source and target node, but also some additional *parameter value* p related to these attributes. For each possible value of p, we can map our flexible graph to a static graph $G_p=(V_p,E_p)$, such that for each node pair $s,t\in V$ there exist

[1] Linking is similar to function composition: $g*f$ means g "after" f.



(a) flexible graph G (b) static graph $G_{\langle 1,3\rangle}$

Figure 2.3: Flexible graph with two edge weight functions. An edge e is labeled with $\left\langle c^{(1)}(e), c^{(2)}(e)\right\rangle$. For the fixed coefficient vector $\langle 1,3\rangle$, a static graph is shown.

$s_p, t_p \in V_p$ such that there is a bijection between the shortest paths between s and t with parameter p in G and the shortest paths between s_p and t_p in G_p. We denote the shortest-path distance in dependence of p with $\mu_p(s,t)$.

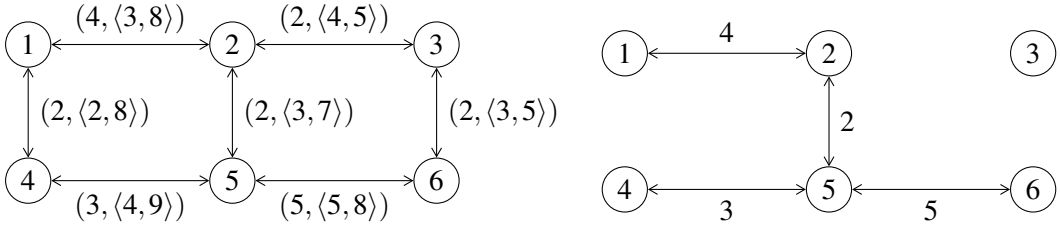
Multiple edge weights. Our additional attributes are r edge weights $c^{(1)}, \ldots, c^{(r)} : E \to \mathbb{R}$ that are linearly combined to a single non-negative real-valued edge weight using the coefficient vector $p = \langle p_1, \ldots, p_r \rangle$. A shortest path for these parameters is a shortest path in the static scenario using the graph $G_p = (V,E)$ with the edge weight function $c_p : e \mapsto \sum_{i=1}^{r} p_i \cdot c^{(i)}(e)$. See Figure 2.3 for an example. So Lemma 2.1 follows directly.

Lemma 2.1 *Let G be a flexible graph with multiple edge weights. The shortest paths in G from source s to target t with parameter value p are exactly the shortest paths in G_p with edge weight function c_p.*

Edge restrictions. We have an edge weight function $c : E \to \mathbb{R}_+$ and an r-dimensional *threshold* function vector $a = \langle a_1, \ldots, a_r \rangle$ such that $a_i : E \to \mathbb{R}_+ \cup \{\infty\}$. Our query parameter is a *constraint* vector $p = \langle p_1, \ldots, p_r \rangle$. A shortest path for p is a shortest path in the static scenario using the graph $G_p = (V, E_p)$ with $E_p := \{e \in E \mid \forall i \in \{1, \ldots, r\} : p_i \leq a_i(e)\}$, see Figure 2.4. So Lemma 2.2 follows directly.

Intuitively, you can think of a_i as the height restriction on a road, and of p_i as the height of the vehicle. In particular, it is also possible to model binary restrictions by setting $a_i(e) = 0$ if the edge is restricted, and $a_i(e) = \infty$ otherwise. A query that wants to avoid such restricted edges, just sets $p_i = \infty$.

Lemma 2.2 *Let G be a flexible graph with edge restrictions. The shortest paths in G from source s to target t with parameter value p are exactly the shortest paths in G_p.*



(a) flexible graph G (b) static graph $G_{\langle 3,7\rangle}$

Figure 2.4: Flexible graph with two restricting threshold functions. An edge e is labeled with $(c(e), \langle a_1(e), a_2(e)\rangle)$. To enforce restrictions $\langle 3,7\rangle$, violating edges are removed in the static graph.

2.3 Public Transportation Networks

Traditionally, a timetable is represented by a set of trains (or buses, ferries, etc). Each train visits a sequence of stations (or bus stops, ports, etc). For each station, except the last one, the timetable includes a *departure time*, and for each station, except the first one, the timetable includes an *arrival time*, see Table 2.5.

Table 2.5: Traditional timetable of three trains.

(a) train 1

station		time
A	dep.	8:05
B	arr.	9:55
	dep.	10:02
C	arr.	11:57
	dep.	12:00
D	arr.	13:20

(b) train 2

station		time
C	dep.	12:00
E	arr.	13:00

(c) train 3

station		time
C	dep.	13:00
E	arr.	14:00

To be able to mathematically define connections consisting of several trains, we split them into *elementary connections* [121]. More formally, we are given a set of stations $\mathscr{B}$, a set of *stop events* $\mathscr{Z}_S$ per station $S \in \mathscr{B}$, and a set of *elementary connections* $\mathscr{C}$, whose elements c are 6-tuples of the form $c = (Z_\mathrm{d}, Z_\mathrm{a}, S_\mathrm{d}, S_\mathrm{a}, \tau_\mathrm{d}, \tau_\mathrm{a})$. Such a tuple (elementary connection) is interpreted as train that leaves station S_d at time τ_d after stop Z_d and the *immediately next* stop is Z_a at station S_a at time τ_a, see Table 2.6 If x denotes a tuple's field, then the notation of $x(c)$ specifies the value of x in the elementary connection c. A stop event is similar to a train identifier, but we will show in Table 2.9 that it is slightly more complicated. We define a stop event to be the consecutive arrival and departure of a train at a station, where no transfer is required. For the corresponding arriving elementary connection c_1 and the departing one c_2 holds $Z_\mathrm{a}(c_1) = Z_\mathrm{d}(c_2)$. Furthermore,

a stop event is local to each station, see Table 2.7. We introduce additional stop events for the begin (no arrival) and the end (no departure) of a train.

Table 2.6: Elementary connections of the timetable in Table 2.5.

(a) train 1

(	Z_d,	Z_a,	S_d,	S_a,	τ_d,	τ_a	)
(	1,	1,	A,	B,	8:05,	9:55	)
(	1,	1,	B,	C,	10:02,	11:57	)
(	1,	1,	C,	D,	12:00,	13:20	)

(b) train 2

(	Z_d,	Z_a,	S_d,	S_a,	τ_d,	τ_a	)
(	2,	1,	C,	E,	12:00,	13:00	)

(c) train 3

(	Z_d,	Z_a,	S_d,	S_a,	τ_d,	τ_a	)
(	3,	2,	C,	E,	13:00,	14:00	)

Table 2.7: Stops are local to a station, and map to trains with arrival and departure time. The stops in this example are taken from the elementary connections in Table 2.6.

(a) station C

stop	train	τ_a	τ_d
1	1	11:57	12:00
2	2	-	12:00
3	3	-	13:00

(b) station E

stop	train	τ_a	τ_d
1	2	13:00	-
2	3	14:00	-

The *duration* of an elementary connection c, denoted by $d(c)$, is $\tau_a(c) - \tau_d(c)$.

At a station $S \in \mathscr{B}$, it is possible to *transfer* from one train to another, if the time between the arrival and the departure at the station S is larger than or equal to a given, station-specific, *minimum transfer duration*, denoted by transfer(S).

Let $P = (c_1, \ldots, c_k)$ be a sequence of elementary connections. Define $\text{dep}_i(P) := \tau_d(c_i)$, $\text{arr}_i(P) := \tau_a(c_i)$, $S_d(P) := S_d(c_1)$, $S_a(P) := S_a(c_k)$, $Z_d(P) := Z_d(c_1)$, $Z_a(P) := Z_a(c_k)$, $\text{dep}(P) := \text{dep}_1(P)$, $\text{arr}(P) := \text{arr}_k(P)$, and $d(P) := \text{arr}(P) - \text{dep}(P)$. Such a sequence P is called a *consistent connection* from station $S_d(P)$ to $S_a(P)$ if it fulfills the following two consistency conditions:

1. The departure station of c_{i+1} is the arrival station of c_i.
2. The minimum transfer durations are respected; either $Z_d(c_{i+1}) = Z_a(c_i)$ or $dep_{i+1}(P) - arr_i(P) \geq \text{transfer}(S_a(c_i))$.

Table 2.8: A consistent connection $P = (c_1, c_2, c_3)$ formed by three elementary connections of Table 2.6. The connection has one transfer at station C. Assume a transfer duration at station C of 5 minutes. It would not be consistent to replace c_3 with train 2 that arrives at $arr_3(P) = 12{:}00$ since there are only $3 < 5 = \text{transfer(C)}$ minutes between the arrival of train 1 and the departure of train 2 at station C.

c_i	train	(	Z_d,	Z_a,	S_d,	S_a,	τ_d,	τ_a	)	dep_i	arr_i
c_1	1	(	1,	1,	A,	B,	8:05,	9:55	)	8:05	9:55
c_2	1	(	1,	1,	B,	C,	1:02,	2:57	)	10:02	11:57
c_3	3	(	3,	2,	C,	E,	13:00,	14:00	)	13:00	14:00

Table 2.9: Example of a train that visits a station more than once. Note that station B is visited twice. If we would only store the train with each of its elementary connections, we could create a consistent connection from elementary connections c_1 and c_4, independent of the minimum transfer duration at station B. Although we would arrive at station D at 12:05 when we board at station A at 12:00, the connection does not describe the correct sequence of elementary connections. The two elementary connections, c_2 and c_3, from station B to station C and back to station B are missing.

(a) traditional timetable

station		time
A	dep.	12:00
B	arr.	12:01
	dep.	12:01
C	arr.	12:02
	dep.	12:03
B	arr.	12:04
	dep.	12:04
D	arr.	12:05

(b) elementary connections

c_i	(	Z_d,	Z_a,	S_d,	S_a,	τ_d,	τ_a	)
c_1	(	1,	1,	A,	B,	12:00,	12:01	)
c_2	(	1,	1,	B,	C,	12:01,	12:02	)
c_3	(	1,	2,	C,	B,	12:03,	12:04	)
c_4	(	2,	1,	B,	D,	12:04,	12:05	)

(c) stops at station B

stop	train	τ_a	τ_d
1	1	12:01	12:01
2	1	12:04	12:04

We illustrate the difference between a consistent and inconsistent connection in Table 2.8.

After we have introduced elementary connections and explained how to create connections from them, we are now ready to explain why we use stop events instead of a single train identifier per elementary connection. In fact, previous publications [121] only use a single train identifier, but they lose information from the traditional timetable. The reason is explained in Table 2.9.

2.3.1 Time-dependent Graph

To create a graph from a timetable using the time-dependent model (Section 1.3.2), we first need to compute all train routes. Each station S has a *transfer node* St. For each

train route R, we add to each station S on this route one *train-route node* SrR, and edges, connecting these nodes in the order of the train route. Each of this edges has an assigned travel time function stemming from all the elementary connections that are represented by this edge. We add an edge from St to SrR and one in the other direction. The edge to the transfer node carries the transfer cost including the minimum transfer duration, the other edge has cost zero. We visualize it in Figure 2.10.

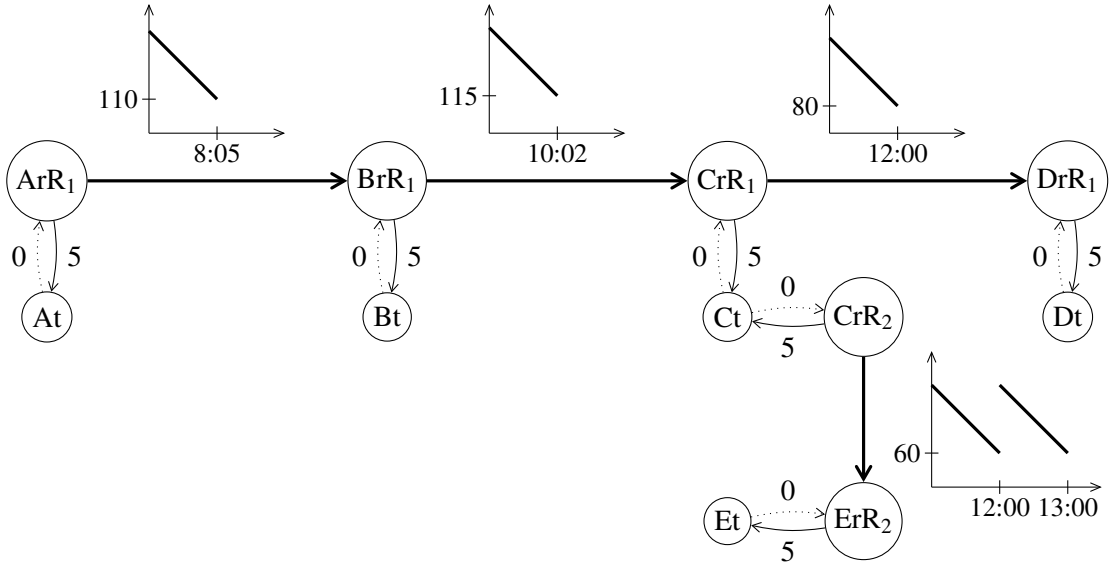


Figure 2.10: Graph in the time-dependent model of the timetable of Table 2.5. There are two train routes, one from station A via B and C to D, and a second one with two connections from station C to E. Each transit edge between two train-route nodes stores a travel time function that maps the departure time to the travel time. The constant slope of the functions represents the waiting for the train. The solid edges between a train-route and the transfer node ensure that the minimum transfer duration is respected. In this example, we assume that the minimum transfer duration is 5 minutes at every station. The dotted edges have cost zero.

2.3.2 Time-expanded Graph

The graph of the time-expanded model has three kinds of nodes, each carries a time and belongs to a station. For every elementary connection $c_1 = (Z_1, Z_2, S_1, S_2, \tau_1, \tau_2)$ from station S_1 to the next station S_2 on the same train, we put a *departure node* $S_1\mathrm{d}@\tau_1$ at S_1 with the departure time τ_1, an *arrival node* $S_2\mathrm{a}@\tau_2$ at S_2 with the arrival time τ_2 and an edge $S_1\mathrm{d}@\tau_1 \to S_2\mathrm{a}@\tau_2$ to model riding this vehicle from S_1 to S_2. If the vehicle continues from S_2 at time τ_3, we put an edge $S_2\mathrm{a}@\tau_2 \to S_2\mathrm{d}@\tau_3$ that represents staying on the vehicle at S_2. This is possible no matter how small the difference $\tau_3 - \tau_2$ is.

For each departure node $S_2\mathrm{d}@\tau$ we put a *transfer node* $S_2\mathrm{t}@\tau$ at the same time and an edge $S_2\mathrm{t}@\tau \to S_2\mathrm{d}@\tau$ between them. Also, we put an edge $S_2\mathrm{t}@\tau \to S_2\mathrm{t}@\tau'$ to the transfer node at S_2 that comes next in the ascending order of departure times (with ties broken arbitrarily); these edges form the *waiting chain* at S_2. Now, to allow a transfer after having reached $S_2\mathrm{a}@\tau_2$, we put an edge to the first transfer node $S_2\mathrm{t}@\tau$ with $\tau \geq \tau_2 + \mathrm{transfer}(S_2)$. This gives the opportunity to transfer to that and all later departures from S_2. We visualize it in Figure 2.11.

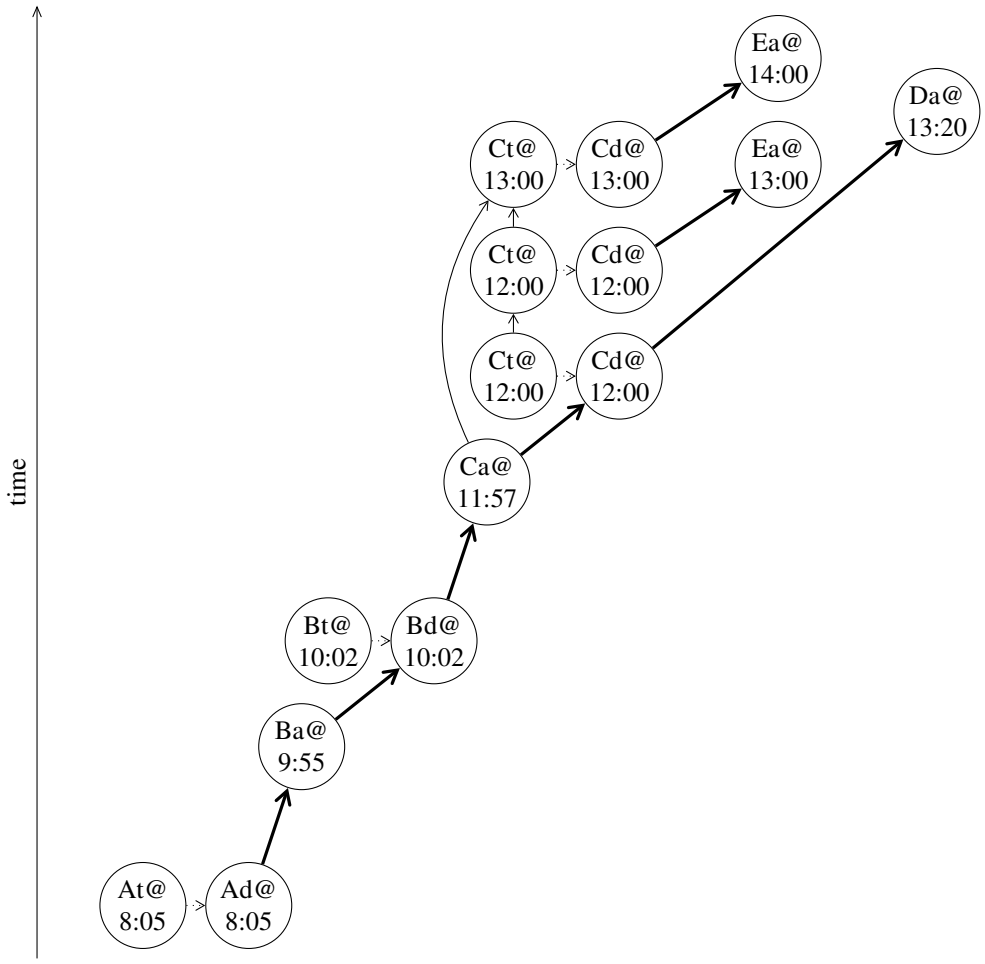


Figure 2.11: Graph in the time-expanded model of the timetable of Table 2.5. For each event, there is a node. We arranged the nodes vertically by there time. The time cost of an edge is implicitly given by the difference between its endpoints. In this example, we assume that the minimum transfer duration is 5 minutes at every station. Therefore, there is an edge from Ca@11:57 to Ct@13:00 and not Ct@12:00.

Multi-criteria. Up to now, we only considered the time in the time tables. However, in the fully realistic scenario, we also support multi-criteria costs. As we use only

time-expanded graphs in the fully realistic scenario, it is sufficient to explain how to incorporate them there. Our scheme supports a fairly general class of multi-criteria cost functions and optimality notions. In our implementation, a cost is a pair (d, p) of non-negative *duration* and *penalty*. Penalty applies mostly to transfers: each station S defines a fixed penalty score for transferring, and that is the penalty component of the cost of edges $S\text{a}@\tau \to S\text{t}@\tau'$. The edges from departure to arrival nodes may be given a small penalty score for using that elementary connection. Other edges, in particular waiting edges, have penalty zero. The cost of a path in the graph is the component-wise sum of the costs of the edges.

We say cost (d_1, p_1) *dominates* or *is better than* cost (d_2, p_2) *in the Pareto sense* iff $d_1 \leq d_2$ and $p_1 \leq p_2$ and one of the inequalities is strict. Each finite set of costs has a unique subset of *Pareto-optimal* costs that are pairwise non-dominating but dominate any other cost in the set (in the Pareto sense).

Optimizations. For exposition, we regard the graph as fully time-expanded, meaning times increase unbounded from time 0 (midnight of day 0). This abstracts from technicalities such as edges that cross midnight and timetables that vary from one day to the next. In practice, we use a more compact graph model that we will describe in Section 4.2.8. Also, we allow additional transfers by walking to nearby stations. More details are provided in Section 4.2.5.

3

Basic Concepts

All of our algorithms use at least one of the basic concepts introduced in this chapter. We will introduce the concepts here for the basic scenario with static graphs (Section 2.2.1). Their augmentation to advanced scenarios is part of the matter of the subsequent chapters. Figure 3.1 gives an overview of the relations.

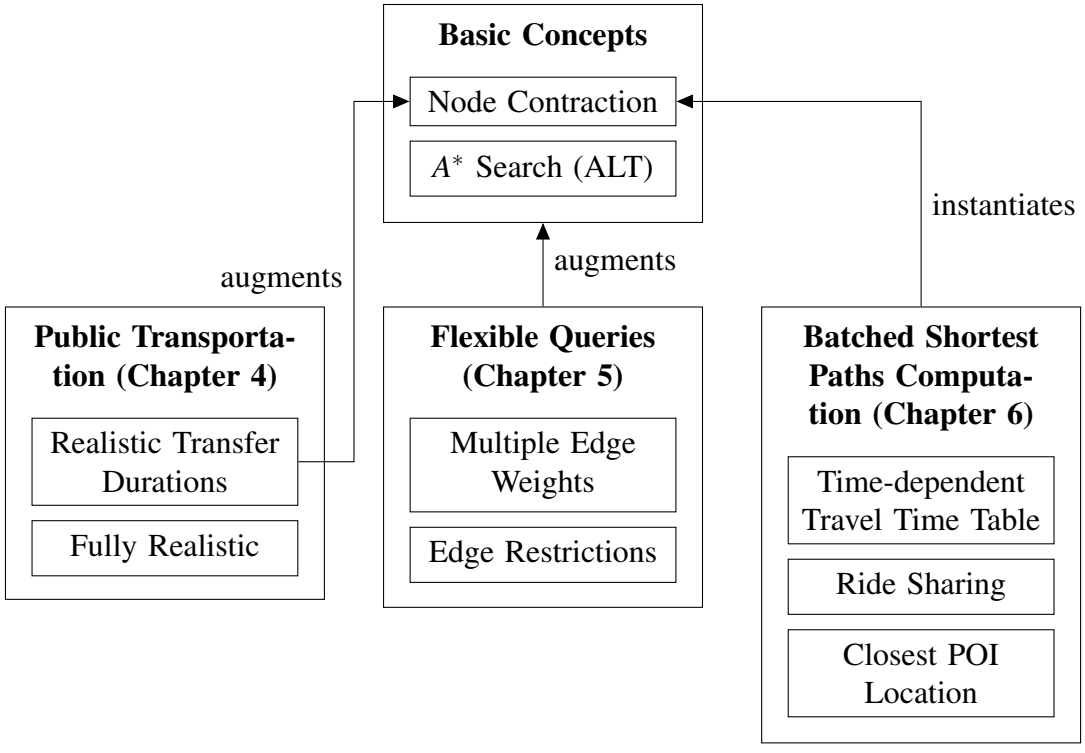


Figure 3.1: Overview of the relations between the basic concepts and the advanced route planning algorithms presented in this thesis. (Note that the relations of Dijkstra's algorithm are omitted as it is related to all other algorithms.)

3.1 Dijkstra's Algorithm

In Chapter 2 we introduced graph models for different scenarios and defined shortest paths. The classical algorithm to compute these shortest paths in a static graph $G = (V, E)$ with edge weight function[1] c is Dijkstra's algorithm [50]. In particular, Dijkstra's algorithm solves the *single-source shortest-path (SSSP) problem* of computing the shortest paths from a single source node s to all other nodes in the graph. The algorithm maintains, for each node u, a label $\delta(u)$ with the tentative distance from s to u. Each node is *unreached*, *reached* or *settled*, see Figure 3.2. A priority queue with key $\delta(u)$ contains all reached but not settled nodes u. Initially, only the source node s is reached with $\delta(s) = 0$ and all other nodes u are unreached with $\delta(u) = \infty$. In each step, a node u in the priority queue with smallest key $\delta(u)$ is deleted. We say that we *settle* node u as in this case we know that $\delta(u)$ is the shortest-path distance. All outgoing edges (u, v) of a settled node u are *relaxed*, i. e., we compare the shortest-path distance from s via u to v with the tentative distance $\delta(v)$. If the one via v is shorter, we update $\delta(v)$ and v in the priority queue. Note that such an update is either an insert operation if v was unreached, or otherwise a decrease key operation. The algorithm terminates once the priority queue is empty, this happens after at most n steps, where n is the number of nodes in the graph. After the termination, all nodes are either unreached or settled.

Algorithm 3.1 shows pseudo-code. Sometimes we do not settle all nodes in the graph, either because there are not all reachable or we stop the search early. To avoid the $O(n)$ initialization in Line 1 for subsequent executions of the algorithm, we can store the settled nodes and reset the tentative distances to ∞ after we used the computed distances. That way, the complexity of the algorithm only depends on the number of settled nodes and relaxed edges, and *not* on n.

Algorithm 3.1: Dijkstra(s)

```
   input  : source s
   output : shortest-path distances δ from s to all nodes in the graph
1  δ := ⟨∞, ..., ∞⟩;                                  // tentative distances
2  δ(s) := 0;                                         // search starts at node s
3  Q.update(0, s);                                    // priority queue
4  while Q ≠ ∅ do
5      (·, u) := Q.deleteMin();                       // settle u
6      foreach e = (u, v) ∈ E do                      // relax edges
7          if δ(u) + c(e) < δ(v) then                 // shorter path via u?
8              δ(v) := δ(u) + c(e);                   // update tentative distance
9              Q.update(δ(v), v);                     // update priority queue
```

Dijkstra's algorithm involves at most n insert operations into the priority queue, n delete operations, and m decrease key operations, yielding a runtime complexity of

[1] We assume non-negative edge weights in static graphs. For arbitrary edge weights, the Bellman-Ford algorithm [24, 57] can be used.

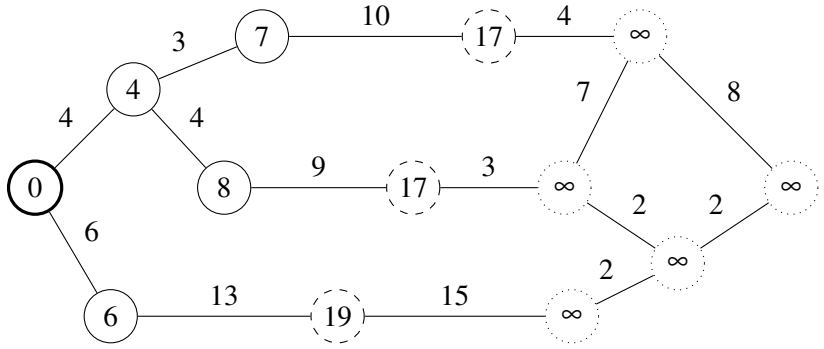


Figure 3.2: Illustration of Dijkstra's algorithm after 5 nodes have been settled. The source node is thick. Nodes are settled (solid), reached (dashed) or unreached (dotted). We labelled each node with its current tentative distance. Each edge is labelled with its weight.

$O(m + n \log n)$ if Fibonacci heaps [133] are used.

There are some simple enhancements of the above algorithm.

Shortest Paths. We can also compute a shortest-path tree using parent pointers. The parent pointer of a node v points to the node u that was relaxed to update v. So a shortest path from s to v can be found by traversing the parent pointers from v until s is reached. As there are potentially multiple shortest paths, we can extend this approach by using sets of parent pointers.

Set of Target Nodes. If only the shortest-path distance to a subset of all nodes should be computed, we can stop the algorithm as soon as all nodes in the subset are settled.

Point-to-Point Queries. If we are only interested in the shortest-path distance to a single target node t, we can use a *bidirectional* version of Dijkstra's algorithm. We execute two searches, one from the source node s in *forward* direction, the other one from the target node t in *backward* direction. The direction defines how edges are relaxed: when we settle a node u in backward direction, we relax all edges (v,u). We can stop the search as soon as a node is settled in both directions. The shortest-path distance is not necessarily the one via this node, but we can guarantee that it is the computed distance from s to t via a node reached in both directions [35].

3.2 Node Contraction

The central idea of node contraction is to remove a node from a graph, and add *shortcut* edges to preserve shortest-path distances between the *remaining* nodes. This can

speedup the shortest path computation by a modified bidirectional Dijkstra search that only relaxes edges to nodes that are contracted later than the currently settled node. When both search scopes meet, a tentative shortest-path distance is updated, and the search is stopped as soon as both minimum keys in the priority queues are not smaller than this tentative distance. The correctness of this approach is ensured by the shortcuts. Highway hierarchies (HH) [123, 124] was the first algorithm using shortcuts, and contraction hierarchies (CH) [67] was the first algorithm only relying on a (more sophisticated) node contraction. The intuition behind node contraction is to contract "unimportant" nodes so that the shortest path search only needs to relax edges to "more important" nodes. While it is possible to contract adjacent nodes at the same time [130], currently the most efficient algorithms contract nodes such that for every edge one endpoint was contracted later, and a total *node order* can be established. We will restrict our explanations to such algorithms.

3.2.1 Preprocessing

In general, the computation of an optimal node order that minimizes the number of necessary shortcuts, or the number of settled nodes in a query, is NP-hard [32, 19]. Therefore, we rely on heuristics. The computation of the node order and the node contraction are combined. We assign each remaining node a *priority* on how attractive it is to contract the node next. Then we contract one of the most attractive nodes and update the priorities of the remaining nodes, as they may depend on contracted nodes or newly added shortcuts. Algorithm 3.2 shows pseudo-code for this simplified preprocessing.

Algorithm 3.2: SimplifiedCHPreprocessing($G = (V,E)$, node order)

```
   input  : graph G
   output : contraction hierarchy G with shortcuts
1  initialize node priorities;
2  while exists remaining nodes do
3      pick one node v with most attractive priority;
4      foreach (u,v) ∈ E with u not contracted, (v,w) ∈ E with w not contracted do
5          if ⟨u,v,w⟩ "may be" the only shortest path from u to w using only uncontracted
           nodes then
               // add shortcut
6              E := E ∪ {(u,w)} (use weight c(u,w) := c(u,v) + c(v,w));
7      update priorities of "some of" the remaining nodes;
```

Adding Shortcuts. To determine the necessary shortcuts, it is sufficient to preserve the shortest-path distances between the neighbors of the currently contracted node v. A simple algorithm is to perform a Dijkstra search from each node u with incoming edge

(u,v) in the remaining graph ignoring node v. The search can stop as soon as all nodes w with outgoing edge (v,w) are settled. If the computed shortest-path distance from node u to a node w is larger than the length of the path $\langle u,v,w\rangle$, then we add a shortcut. Otherwise, there is a *witness* path from u to w that can replace the path via v. We call the Dijkstra search therefore *witness search*. An important observation is that we can prune this witness search, for example by limiting the number of settled nodes, or the depth of the shortest-path tree (*hop limit*). Such a pruned search is called a *local search*, as it usually explores only a small part of the graph. Local searches compute upper bounds on the shortest-path distance from u to w and we may add more shortcuts that necessary. Nevertheless, we always add all necessary shortcuts. To remove unnecessary shortcuts later, we use the information gathered by the witness search to perform an *edge reduction*. More precisely, after a witness search from a node u, we drop each remaining edge (u,x) that has been traversed, who is not the path computed to x. This preserves correctness, as (u,x) on a shortest path could then be replaced by the computed path.

Node Priority. A crucial but also complex part is the computation of the node priorities [59]. Usually, it is a linear combination of certain priority terms considering the number of necessary shortcuts, the cost of the witness search, the (estimated) cost of queries, and uniformly distributed contraction of the nodes. Below is a list of good priority terms.

- The *edge difference* is the difference between the number of incident edges of a node and the number of necessary shortcuts to contract it. It can also be used as the *edge quotient*, the quotient between the two numbers of edges.
- the number of adjacent *contracted neighbors*
- the number of contracted nodes that are closer to this node than to any other node (shortest-path-based *Voronoi* region)
- the *search space size* counting the number of relaxed edges or settled nodes in the witness searches
- the sum of number of *original edges* that are represented by the necessary shortcuts
- The *hierarchy depth* is an upper bound on the depth of a shortest path tree during a query. Initially, the hierarchy depth of each node is 0. When we contract a node, the hierarchy depth of its remaining neighbors is updated by 1 plus the hierarchy depth of the contracted node, if it is not already higher.
- based on some *global measures*, such as betweenness [58, 5] or reach [73]

The selection of the priority terms and the coefficients for the linear combination depend on the advanced scenario, and also on the used graph. Further terms can evolve in specific scenarios, for example in time-dependent road networks (Section 2.2.2), the complexity of the travel time functions is important [15].

Updating Node Priorities. The contraction of a node mostly affects the remaining neighbors of this node, but generally more nodes are affected. As it is not practical to update the node priorities of all remaining nodes, we only update the neighbors. Furthermore, before we contract a node, we recompute its node priority and reconsider our decision to contract it (*layz update*).

3.2.2 Query

Node contraction creates the structure described by Lemma 3.1 on the processed graph by adding shortcuts. These shortcuts represent whole paths in the original graph and therefore do not change the shortest-path distances. But the shortcuts may add additional representations of shortest paths. We exploit this with our query algorithm.

Lemma 3.1 *Let s be a source node and t be a target node that are connected and potentially already contracted. There exists a shortest s-t-path P in the graph including shortcuts of the form $\langle s,\dots,x,\dots,y,\dots,t\rangle$ with the following properties:*

(1) If $s \neq x$, then the subpath $\langle s,\dots,x\rangle$ is an upward path*: Each node, starting with s, was contracted before the next one on the path. Node x may not be contracted, but all other nodes are.*

(2) If $y \neq t$, then the subpath $\langle y,\dots,t\rangle$ is a downward path*: Each node, starting with t, was contracted before the previous one on the path. Node y may not be contracted, but all other nodes are.*

(3) If $x \neq y$, then all nodes on the subpath $\langle x,\dots,y\rangle$ are in the remaining graph, and thus not contracted.

Proof. This proof is an extended version of the correctness proof for contraction hierarchies [59, Theorem 2]. Let s be a source node and t be a target node. The contraction of a node preserves the shortest-path distances between the remaining neighbors by adding shortcuts. For a path P, among all contracted nodes v on P whose two neighbors on P are not contracted earlier, define node z_P being the one that was contracted the earliest, or $z_P = \perp$ if no such node exists. The two neighbors may not be contracted at all. Choose among all shortest s-t paths the path P with $z_P = \perp$ or z_P being contracted the latest. We will prove that P has the required form.

If s has been contracted, choose node x such that the subpath $\langle s,\dots,x\rangle$ is the maximal upward path, otherwise $x := s$. And if t has been contracted, choose node y such that the subpath $\langle y,\dots,t\rangle$ is the maximal downward path, otherwise $y := t$. Therefore, the path $\langle s,\dots,x,\dots,y,\dots,t\rangle$ has properties (1) and (2), and it remains to show property (3). Assume that $x \neq y$, and for the sake of contradiction, that a node v on the subpath $\langle x,\dots,y\rangle$ has been contracted. Then, there is also such a node v that has been contracted before x and y, as the subpaths $\langle s,\dots,x\rangle$ and $\langle y,\dots,t\rangle$ are maximal. Therefore holds

$z_P \neq \perp$. The contraction of z_P either adds a shortcut between the neighbors, or there exists a witness consisting of nodes not being contracted earlier than z_P. This results in a shortest s-t-path P' with $z_{P'} = \perp$ or $z_{P'}$ is contracted later than z_P, contrary to our selection of P. □

Corollary 3.2 *Assume that all nodes in the graph are contracted. Then for each connected source node s and target node t, there is a shortest path of the form $\langle s,\dots,x,\dots,t\rangle$. The subpath $\langle s,\dots,x\rangle$ is an upward path, and the subpath $\langle x,\dots,t\rangle$ is a downward path. Such a path is called an* up-down path.

Corollary 3.3 *For each connected source node s and target node t, that are not contracted, there is a shortest path between them consisting only of uncontracted nodes.*

Here, we assume that our preprocessing contracts *all* nodes. We will discuss the case where we do not contract all nodes in Section 3.4. After the preprocessing, we split the resulting graph with shortcuts into an *upward graph* $\overrightarrow{G} := (V, \overrightarrow{E})$ with $\overrightarrow{E} := \{(u,v) \in E \mid u \text{ contracted before } v\}$ and a *downward graph* $\overleftarrow{G}$:= $(V, \overleftarrow{E})$ with $\overleftarrow{E} := \{(u,v) \in E \mid u \text{ contracted after } v\}$.

Our query algorithm simultaneously performs a forward search in $\overrightarrow{G}$ and a backward search in $\overleftarrow{G}$. Both search scopes will meet at the most important node of a shortest path, as Corollary 3.2 ensures. An important observation is that we only need to store an edge in the edge group of the less important incident node. This formally results in a *search graph* $G^* = (V, E^*)$ with $\overrightarrow{\overleftarrow{E}} := \left\{(v,u) \mid (u,v) \in \overleftarrow{E}\right\}$ and $E^* := \overrightarrow{E} \cup \overrightarrow{\overleftarrow{E}}$. And we store a forward and a backward flag such that for any edge $e \in E^*$, $\rightarrow(e) =$ true iff $e \in \overrightarrow{E}$ and $\leftarrow(e) =$ true iff $e \in \overrightarrow{\overleftarrow{E}}$. Algorithm 3.3 presents pseudo-code for a query in G^*.

Stall-on-demand. An important optimization technique is stall-on-demand [130]. It exploits the fact that nodes are settled with a suboptimal distance that is larger then the shortest-path distance. This happens as we do not relax all edges compared to Dijkstra's algorithm. We detect this by looking at the edges that are used by the search in the other direction to build a path to the currently settled node from an more important neighbor. If this path is shorter, we *stall* the currently settled node, meaning we do not relax its edges. This is correct, as this node would never be part of a shortest path that is found by our query algorithm. Furthermore, we can propagate this path to other nodes to stall more suboptimally reached nodes. The most efficient way proved to be just the propagation to its reached neighbors. When those nodes are settled later without being updated in between, we stall them.

Algorithm 3.3: CHQuery(s, t)

input : source s, target t
output : shortest-path distance δ

$\overrightarrow{\delta} := \langle \infty, \ldots, \infty \rangle$; // tentative forward distances
$\overleftarrow{\delta} := \langle \infty, \ldots, \infty \rangle$; // tentative backward distances
$\overrightarrow{\delta}(s) := 0$; // forward search starts at node s
$\overleftarrow{\delta}(t) := 0$; // backward search starts at node t
$\delta := \infty$; // tentative shortest-path distance
$\overrightarrow{Q}$.update(0, s); // forward priority queue
$\overleftarrow{Q}$.update(0, t); // backward priority queue
$\sim := \rightarrow$; // current direction
while $(\overrightarrow{Q} \neq \emptyset)$ **or** $(\overleftarrow{Q} \neq \emptyset)$ **do**
  if $\delta < \min\left\{\overrightarrow{Q}.\min(), \overleftarrow{Q}.\min()\right\}$ **then break**;
  if $\widetilde{Q} \neq \emptyset$ **then** $\sim := \neg \sim$; // change direction, $\neg \leftarrow = \rightarrow$ and $\neg \rightarrow = \leftarrow$
  $(\cdot, u) := \widetilde{Q}$.deleteMin(); // u is settled
  $\delta := \min\left\{\delta, \overrightarrow{\delta}(u) + \overleftarrow{\delta}(u)\right\}$; // u is potential candidate
  foreach $e = (u, v) \in E^*$ **with** $\sim(e)$ **do** // relax edges
    if $\left(\widetilde{\delta}(u) + c(e)\right) < \widetilde{\delta}(v)$ **then** // shorter path via u?
      $\widetilde{\delta}(v) := \widetilde{\delta}(u) + c(e)$; // update tentative distance
      $\widetilde{Q}$.update$\left(\widetilde{\delta}(v), v\right)$; // update priority queue
return δ;

3.3 A* Search

A^* search [77] is a goal-directed technique that helps to speed up Dijkstra's algorithm by pushing the search towards the target. Given a (target-dependent) potential $\pi : V \to \mathbb{R}$, A^* search is a Dijkstra search executed on the reduced edge weights $c_\pi(u,v) = c(u,v) - \pi(u) + \pi(v)$. A potential is *feasible* iff $c_\pi(u,v) \geq 0$, a necessary condition for Dijkstra's algorithm. So, if the potential $\pi(t)$ of the target t is zero, $\pi(v)$ is lower bound on the shortest-path distance from v.

3.3.1 Landmarks (ALT)

The ALT algorithm [69, 72] based on **A***, **L**andmarks and the **T**riangle inequality, is currently the most efficient approach to compute good potentials for road networks. Given a set of landmarks $L \subset V$, we precompute all shortest-path distances from and to these landmarks. With the triangle inequality, we can derive a lower bound from an arbitrary node v to an arbitrary target node t. Let $\ell \in L$, then $\mu(v,t) \geq \mu(v,\ell) - \mu(t,\ell)$ and

$\mu(v,t) \geq \mu(\ell,t) - \mu(\ell,v)$. Each of these lower bounds, and also the maximum over all landmarks is a feasible potential.

Landmark Selection. The most commonly used method to select landmarks is the *avoid* heuristic [69]. It provides very good potentials while still having fast precomputation time. In iteratively creates a set of landmarks. It grows a shortest-path tree from a random node r in backward direction using Dijkstra's algorithm. The *size* of a node v in the tree is recursively defined by the difference between $\mu(v,r)$ and the potential obtained from the current set of landmarks, plus the size of its children. We set the size of all nodes v to zero that have at least one landmark in the subtree rooted at v. Then, starting at the node with highest size, we follow the child with the highest size until we reach a leaf. This leaf is added to the set of landmarks. The first random node r is picked uniformly at random, the following ones picked with a probability proportional to the square of the distance to the nearest landmark. Further landmark selection methods can be found in [69, 72].

3.3.2 Bidirectional A*

While an unidirectional search works with any feasible potential, a bidirectional search needs more caution. In principle, we can execute a search using two potentials $\overrightarrow{\pi}$ and $\overleftarrow{\pi}$ for forward and backward direction. But it is only allowed to stop the search as soon as a node is settled in both directions with a *consistent* pair of potential functions fulfilling $\overrightarrow{\pi} + \overleftarrow{\pi} = \text{const}$.

3.4 Combination of Node Contraction and ALT

As the computation and storage of the distances to landmarks is quite expensive on a large graph, we can combine it with node contraction [23, 38]. We first contract nodes until a *core* of the K most important nodes remains, and then we compute the landmarks for this core. That requires to store the landmark distances only for the core nodes. Then, the bidirectional query can guide its search with A^* within the core.

More precisely, we perform the query in two phases. In the first phase, we perform the CH query described in Section 3.2.2, but stop at core nodes. In the second phase, we perform a core-based bidirectional ALT algorithm. As source node s and target node t are not necessarily in the core, we need *proxy nodes* s' and t' in the core [71]. The ALT potentials are computed using s' and t' as goal, so the best proxy nodes are the core entry and exit node of a shortest path from s to t. However, as our query wants to compute a shortest path and does not know it in advance, we use the core nodes that are closest to s and t as proxy nodes. We split the second phase into two parts itself. The first part is the calculation of the proxy nodes. We perform a backward search from the source node and a forward search from the target node until we settle the first core node. Note, that it is

not guaranteed to find a path to the core in the respective searches. This can only happen if our input graph does not consist of strongly connected components. If this should be the case we disable the goal-direction for the respective direction by setting the potential function to zero. The second part is a bidirectional ALT algorithm, starting from all the core nodes that are settled in the first phase. The path found by this search has the form described by Lemma 3.1.

We can even extend the approach of [23, 38] by contracting the core and applying a bidirectional algorithm on the core that combines the CH and ALT algorithm. This provides faster query times at the expense of slower preprocessing, as the core is usually quite dense and takes long to contract. On a contracted core, due the stopping criterion of the CH query (Section 3.2.2), consistent potential functions will never improve the query, as we are not allowed to stop as soon as both directions meet. The only modification of this query algorithm is that we now add the potential functions to the priority keys.

As the combination of ALT and CH in the core is new, we will look into the details and its correctness. We first consider the case where we have landmarks for the whole graph, such that we have a query algorithm with just one phase. The resulting CHALT[2] query is a slight adaption of the CH query by naturally adding the potential functions, see Algorithm 3.4. Lemma 3.4 proves its correctness. Note that it is does not matter whether the landmark distances are computed before or after the contraction of the core nodes, as the node contraction does not change shortest-path distances, see Corollary 3.3.

Lemma 3.4 *The CHALT query algorithm computes* $\mu(s,t)$.

Proof. As the potentials are feasible, the nodes on the shortest path given by Corollary 3.2 are settled with the same distance as the CH algorithm. Assume for the sake of contradiction, that our query does not return $\mu(s,t)$. This can only happen if we abort the query too early in Line 11 with a path via candidate u, and therefore $\delta > \mu(s,t)$. Assume that node v is an optimal candidate. Therefore, at the time of abortion, $\overrightarrow{\delta}(u) + \overrightarrow{\pi}(u) + \overleftarrow{\pi}(u) + \overleftarrow{\delta}(u) = \delta_\pi < \min\left\{\overrightarrow{Q}.\min(), \overleftarrow{Q}.\min()\right\} \leq \min\left\{\overrightarrow{\delta}(v) + \overrightarrow{\pi}(v), \overleftarrow{\delta}(v) + \overleftarrow{\pi}(v)\right\}$. As $\mu(s,v) + \overrightarrow{\pi}(v) \leq \mu(s,v) + \mu(v,t) + \overrightarrow{\pi}(t) = \mu(s,t) + \overrightarrow{\pi}(t) < \overrightarrow{\delta}(u) + \overleftarrow{\delta}(u) + \overrightarrow{\pi}(u)$, v is settled with $\overrightarrow{\delta}(v) = \mu(s,v)$ in the forward search before the abortion. A symmetric argument gives that v is settled with $\overleftarrow{\delta}(v) = \mu(v,t)$ in the backward search. From that we can follow that $\delta \leq \overrightarrow{\delta}(v) + \overleftarrow{\delta}(v)$ (Line 14), a contradiction to the initial assumption that $\delta > \mu(s,t) = \overrightarrow{\delta}(v) + \overleftarrow{\delta}(v)$. □

A query that uses landmarks on a contracted core, just replaces the bidirectional ALT query in the second phase by our CHALT query, starting from all the core nodes that are settled in the first phase. The proof of correctness (Theorem 3.5) is straightforward.

[2]CHALT combines CH and ALT in a *single* query phase (no core), easily mistakeable for CALT, a commonly used abbreviation for Core-ALT that has *two separate* query phases and uses ALT on an uncontracted core.

Algorithm 3.4: CHALTQuery(s, t)

input : source s, target t
output : shortest-path distance δ

1 $\overrightarrow{\delta} := \langle \infty, \ldots, \infty \rangle$; // tentative forward distances
2 $\overleftarrow{\delta} := \langle \infty, \ldots, \infty \rangle$; // tentative backward distances
3 $\overrightarrow{\delta}(s) := 0$; // forward search starts at node s
4 $\overleftarrow{\delta}(t) := 0$; // backward search starts at node t
5 $\delta := \infty$; // tentative shortest-path distance
6 $\delta_\pi := \infty$; // tentative shortest-path distance with potentials
7 $\overrightarrow{Q}$.update($0 + \overrightarrow{\pi}(s), s$); // forward priority queue
8 $\overleftarrow{Q}$.update($0 + \overleftarrow{\pi}(t), t$); // backward priority queue
9 $\sim := \rightarrow$; // current direction
10 **while** ($\overrightarrow{Q} \neq \emptyset$) **or** ($\overleftarrow{Q} \neq \emptyset$) **do**
11 **if** $\delta_\pi < \min\left\{\overrightarrow{Q}.\min(), \overleftarrow{Q}.\min()\right\}$ **then break**;
12 **if** $\widetilde{Q} \neq \emptyset$ **then** $\sim := \neg \sim$; // interleave direction, $\neg \leftarrow = \rightarrow$ and $\neg \rightarrow = \leftarrow$
13 $(\cdot, u) := \widetilde{Q}$.deleteMin(); // u is settled
14 $\delta_\pi := \min\left\{\delta_\pi, \overrightarrow{\delta}(u) + \overrightarrow{\pi}(u) + \overleftarrow{\pi}(u) + \overleftarrow{\delta}(u)\right\}$; // u is potential candidate
15 $\delta := \min\left\{\delta, \overrightarrow{\delta}(u) + \overleftarrow{\delta}(u)\right\}$;
16 **foreach** $e = (u, v) \in E^*$ **with** $\sim(e)$ **do** // relax edges
17 **if** $\left(\widetilde{\delta}(u) + c(e)\right) < \widetilde{\delta}(v)$ **then** // shorter path via u?
18 $\widetilde{\delta}(v) := \widetilde{\delta}(u) + c(e)$; // update tentative distance
19 $\widetilde{Q}$.update$\left(\widetilde{\delta}(v) + \widetilde{\pi}(v), v\right)$; // update priority queue
20 **return** δ;

Theorem 3.5 *Our two phase query algorithm using CHALT on a contracted core computes* $\mu(s,t)$.

Proof. Let node v be an optimal candidate node where forward and backward search of a CH query can meet to compute $\mu(s,t) = \overrightarrow{\delta}(v) + \overleftarrow{\delta}(v)$. If v is not in the core, the first phase will already compute the correct shortest-path distance. Otherwise, v is in the core and we can apply the proof of Lemma 3.4. □

4

Public Transportation

We have two major contributions in this chapter. First, we use the ideas of a speed-up technique for route planning to create a new algorithm for the scenario with realistic transfer durations. And second, we contribute to a completely new algorithm that can handle the fully realistic scenario.

4.1 Routing with Realistic Transfer Durations

4.1.1 Central Ideas

Hierarchical route planning algorithms are very successful on road networks, but they fail on public transportation networks, especially when based on the concept of node contraction (Section 3.2). Intuitively, for public transportation networks, we want to contract *stations*. However, previous models either use *several nodes per station*, i. e. the time-expanded model [110, 102, 132] or the time-dependent model [29, 111, 116, 117, 121], or require *parallel edges* [26, 25]. In such models, there can be potentially many edges between the same pair of stations, either between different nodes representing the stations, or as parallel edges. This is disadvantageous for node contraction, and we propose to use a new *station graph model*, that represents each station by a single node, and does not require parallel edges. However, this station graph model requires fairly complex edge costs, so that it requires some effort to efficiently implement the basic edge operations required to chain and merge edges. The second central idea is to exploit the small number of stations in a public transportation network, compared to the number of nodes in a road networks. This allows us to *cache* more data during preprocessing, leading to very fast preprocessing times.

4.1.2 Station Graph Model

We introduce a model that represents a timetable with a set of stations $\mathscr{B}$ as a directed graph $G = (\mathscr{B}, E)$ with exactly one node per station. For a simplified scenario without

realistic transfer durations, this is like the time-dependent model (Section 2.3.1). We explain in Example 4.1 why a station graph model is advantageous over the time-dependent model for contraction.

Example 4.1 *Assume that we have a network with three stations A, B and C. Let there be three different train routes*[1] R_1, R_2, *and* R_3 *from station A to station B, and three different train routes* R_4, R_5, *and* R_6 *from station B to station C. As described in Section 2.3.1, these train routes are represented by 3 train-route nodes* $ArR_1, \ldots, ArR_3$ *at station A,* $3+3=6$ *train-route nodes* $BrR_1, \ldots, BrR_6$ *at station B and 3 train-route nodes* $CrR_4, \ldots, CrR_6$ *at station C. To allow transfers at stations, there is a dedicated station node and transfer edges from/to the train-route nodes. The time-dependent graph is shown in Figure 4.1.*

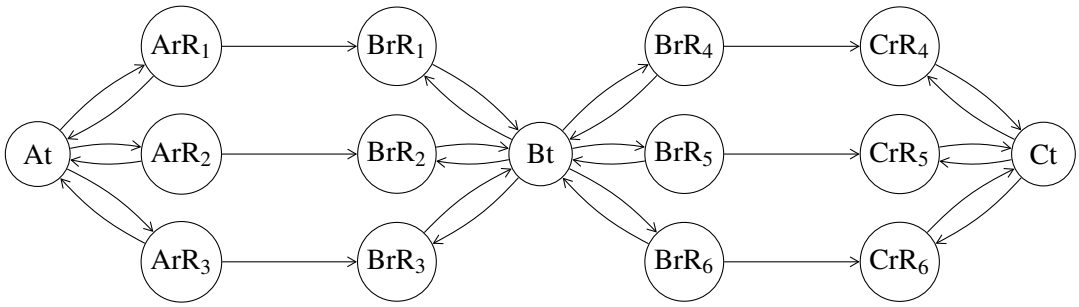


Figure 4.1: Example graph in the time-dependent model before the contraction of station B. There are 3 stations and 6 train routes.

Now assume that we contract station B with all its nodes, and that we always need to add a shortcut. The resulting graph is shown in Figure 4.2. So the initially 3 outgoing edges from the train-route nodes of station A and the 3 incoming edges to the train-route nodes of station C multiplied to $3 \cdot 3 = 9$ *edges.*

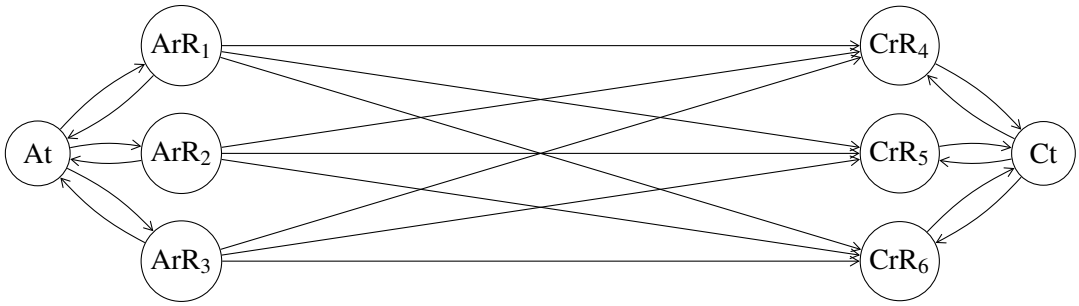
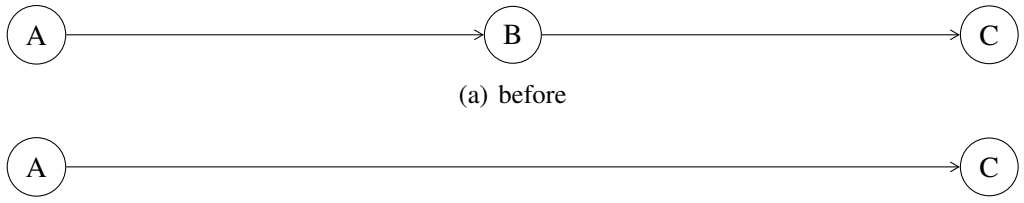


Figure 4.2: Example graph in the time-dependent model. After the contraction of station B, there can be an edge from every train-route node of A to every train-route node of C.

[1] As introduced in Section 1.3.2, a *train route* is a subset of trains that follow the exact same route, at possibly different times and do not overtake each other [121].

Figure 4.3 shows the graph of the same network, but in the station graph model, before and after the contraction. There is just a single node per station and no parallel edges. During the contraction of station B, we just add a single shortcut from station A to station C representing a set of connections (these connections do not *need to have the same train-route and the set also* does not *need to have the FIFO-property).*



(a) before

(b) after

Figure 4.3: Example graph in the station graph model before and after the contraction of station B. There is a single node per station.

In the station graph model, we store for each edge $e = (A, B) \in E$ a set of consistent connections $c(e)$ that depart at A and arrive at B, usually all elementary connections. Here and in the following we assume that all connections are consistent, as defined in Section 2.3. Note that this set of connections $c(e)$ does not need to fulfill the FIFO-property, i. e. some connections may overtake each other. In comparison, the station graph model of Berger et al. [26, 25] partitions the set $c(e)$, such that the set of connections represented by a partition fulfills the FIFO-property. Then, each set of connections represented by a partition is stored as a separate parallel edge. As we want to avoid parallel edges, in general, an edge in our model cannot represent a set of connections with FIFO-property. Without FIFO-property, the relaxation of an edge during a search is more complicated, even for the computation of earliest arrival times (formally introduced by Definition 4.5 in Section 4.1.3). We provide an efficient search algorithm in Section 4.1.5.

We say that a connection P *dominates* a connection Q if we can replace Q by P (Lemma 4.4). More formally, let Q be a connection. Define $\text{parr}(Q)$ as the (**p**revious) **arr**ival time of the train at station $S_\text{d}(Q)$ before it departs at time $\text{dep}(Q)$, or $\perp$ if this train begins there. If $\text{parr}(Q) \neq \perp$ then we call $\text{st}_\text{d}(Q) := \text{dep}(Q) - \text{parr}(Q)$ the **s**topping **t**ime at **d**eparture. We say that Q has a *critical departure* when $\text{parr}(Q) \neq \perp$ and $\text{st}_\text{d}(Q) < \text{transfer}(S_\text{d}(Q))$. Symmetrically, we define $\text{ndep}(Q)$ as the (**n**ext) **dep**arture time of the train at station $S_\text{a}(Q)$, or $\perp$ if the train ends there. When $\text{ndep}(Q) \neq \perp$ then we call $\text{st}_\text{a}(Q) := \text{ndep}(Q) - \text{arr}(Q)$ the **s**topping **t**ime at **a**rrival. And Q has a *critical arrival* when $\text{ndep}(Q) \neq \perp$ and $\text{st}_\text{a}(Q) < \text{transfer}(S_\text{a}(Q))$. We will motivate the definition of a critical departure and arrival in Example 4.3 later, but first we need to define the dominance relation between connections.

Definition 4.2 *A connection P* dominates *Q iff all of the following conditions are fulfilled:*

(1) $S_{\mathrm{d}}(P) = S_{\mathrm{d}}(Q)$ *and* $S_{\mathrm{a}}(P) = S_{\mathrm{a}}(Q)$

(2) $\mathrm{dep}(Q) \leq \mathrm{dep}(P)$ *and* $\mathrm{arr}(P) \leq \mathrm{arr}(Q)$

(3) $Z_{\mathrm{d}}(P) = Z_{\mathrm{d}}(Q)$*, or* Q *does not have a critical departure, or* $\mathrm{dep}(P) - \mathrm{parr}(Q) \geq \mathrm{transfer}(S_{\mathrm{d}}(P))$

(4) $Z_{\mathrm{a}}(P) = Z_{\mathrm{a}}(Q)$*, or* Q *does not have a critical arrival, or* $\mathrm{ndep}(Q) - \mathrm{arr}(P) \geq \mathrm{transfer}(S_{\mathrm{a}}(P))$

Conditions (1) and (2) are elementary conditions. Conditions (3) and (4) are necessary to respect the minimum transfer durations, when Q is a subconnection of a larger connection, see Example 4.3. Given connection $R = (c_1, \ldots, c_k)$ consisting of k elementary connections, we call a connection $(c_i, \ldots, c_j)$ with $1 \leq i \leq j \leq k$ a *subconnection* of R, we call it *prefix* iff $i = 1$ and *suffix* iff $j = k$. Given a set of connections $\mathscr{S}$, we call a subset $\mathscr{S}' \subseteq \mathscr{S}$ *dominant set among* $\mathscr{S}$ iff for all $Q \in \mathscr{S} \setminus \mathscr{S}'$ exists $P \in \mathscr{S}'$ such that P dominates Q, and no connection $P \in \mathscr{S}'$ dominates a connection $Q \in \mathscr{S}' \setminus \{P\}$.

Example 4.3 *This example motivates the notion of a* critical departure/arrival *and Definition 4.2 of the dominance relation on connections. Let our timetable be given by Table 4.4. As there is at most a single elementary connection between each pair of stations, we can represent connections by their sequence of stations.*

Table 4.4: Timetable consisting of two trains and six stations. The minimum transfer duration at every station is 5 minutes.

(a) train 1

station		time
A	dep.	9:00
B	arr.	9:06
	dep.	9:10
C	arr.	9:15
	dep.	9:17
E	arr.	9:20
	dep.	9:24
F	arr.	9:40

(b) train 2

station		time
B	dep.	9:10
D	arr.	9:12
	dep.	9:14
E	arr.	9:19

Let Q be the connection $B \to C \to E$ and let P be the connection $B \to D \to E$. Both connections are obviously consistent, as Q only uses train 1 and P only uses train 2. Some of the attributes of Q and P are summarized in Table 4.5. Let us decide whether connection P dominates connection Q, i. e. we can replace connection Q by P. The conditions (1) and (2) are already fulfilled. Both depart at 9:10 and the duration of Q is 10 minutes and the duration of P is 9 minutes. At first glance, it might look like we can replace Q by P.

connection	parr	dep	st_d	critical departure	arr	ndep	st_a	critical arrival
Q	9:06	9:10	4	yes	9:20	9:24	4	yes
P	$\perp$	9:10	–	no	9:19	$\perp$	–	no

Table 4.5: Attributes of connections Q and P.

However, if Q appears as subconnection of a larger connection, the result may not be consistent. Let R be the connection $A \to B \to C \to E \to F$. Connection R is consistent as it is a single train from A to F without any transfers. Q is a subconnection of R. When we replace Q by P in R, we get a connection R' over $A \to B \to D \to E \to F$ with transfers at B and E. Connection R' is not consistent. The reason is that between the arrival of train 1 at B at 9:06 and the departure of train 2 at 9:10 are only 4 minutes time, this is smaller than the minimum transfer duration of 5 minutes. If R would arrive earlier, e. g. at 9:05, the transfer would be consistent, $\text{st}_\text{d}(Q)$ *would increase to 5 minutes and Q would not have a critical departure.*

Note that at station E we have a different situation than at station B, although Q has a critical arrival (at E). There, the transfer of R' is consistent since between the arrival at 9:19 and the departure at 9:24 are 5 minutes time to transfer. However, in general, conditions (1) and (2) are only sufficient if Q has neither a critical departure nor a critical arrival.

Lemma 4.4 *A consistent connection P dominates a consistent connection Q iff for all consistent connections R with subconnection Q, we can* replace *Q by P to get a consistent connection R' with $S_\text{d}(R) = S_\text{d}(R')$, $S_\text{a}(R) = S_\text{a}(R')$, and* $\text{dep}(R) \leq \text{dep}(R') \leq \text{arr}(R') \leq \text{arr}(R)$.

Proof. $\Rightarrow$: P dominates Q: We need to show that R' is a consistent connection. Condition (1) ensures that $S_\text{d}(R) = S_\text{d}(R')$, $S_\text{a}(R) = S_\text{a}(R')$. Condition (2) ensures that we can replace $R = Q$ by P directly or when transfer durations are irrelevant. The last two conditions (3) and (4) ensure that we can replace Q by P even when Q is just a part of a bigger connection and we need to consider transfer durations. The prefix of this bigger connection w. r. t. Q may arrive in $S_\text{d}(Q)$ with stop event $Z_\text{d}(Q)$ and due to condition (3) it is consistent to transfer to $Z_\text{d}(P)$. Consequently the suffix of this bigger connection w. r. t. Q may depart in $S_\text{a}(Q)$ with stop event $Z_\text{a}(Q)$ and due to condition (4) it is consistent to transfer from $Z_\text{a}(P)$. If Q is not a prefix of R, then $\text{dep}(R) = \text{dep}(R')$, otherwise, condition (2) ensures that $\text{dep}(R) = \text{dep}(Q) \leq \text{dep}(P) = \text{dep}(R')$. If Q is not a suffix of R, then $\text{arr}(R) = \text{arr}(R')$, otherwise, condition (2) ensures that $\text{arr}(R') = \text{arr}(P) \leq \text{arr}(Q) = \text{arr}(R)$.

$\Leftarrow$: $\forall R$ we can replace Q by P: We need to show that all 4 conditions hold to prove that P dominates Q. Condition (1) holds trivially. We get condition (2) with $R = Q$.

Condition (3): Let $Q = (c_1, \ldots, c_k)$ have a critical departure and $Z_\text{d}(Q) \neq Z_\text{d}(P)$. Then there is a previous arrival of the train at station $S_\text{d}(Q)$ with elementary connection

c_0. We choose $R := (c_0, c_1, \ldots, c_k)$. As we can replace Q by P in R to get a consistent connection, $\text{dep}(P) - \text{parr}(Q) \geq \text{transfer}(S_\text{d}(P))$ must hold.

Condition (4): Let $Q = (c_1, \ldots, c_k)$ have a critical arrival and $Z_\text{a}(Q) \neq Z_\text{a}(P)$. Then there is a next departure of the train at station $S_\text{a}(Q)$ with elementary connection c_{k+1}. We choose $R := (c_1, \ldots, c_k, c_{k+1})$. As we can replace Q by P in R to get a consistent connection, $\text{ndep}(Q) - \text{arr}(P) \geq \text{transfer}(S_\text{a}(P))$ must hold. □

4.1.3 Query

We consider two types of queries in the scenario with realistic transfer durations: A *time query* (Definition 4.5) solving the earliest arrival problem[2] (EAP), and a profile query (Definition 4.6) computing a dominant set of connections.

Definition 4.5 *A* time query *is given as $A@\tau \to B$, where A is the source station, τ is the earliest departure time, and B is the target station. All consistent connections from station A to station B that do not depart before τ are* feasible *for this query. A feasible connection with minimal arrival time is* optimal. *The query's result is an optimal connection.*

Definition 4.6 *A* profile query *is given as $A \to B$, where A is the source station, and B is the target station. All consistent connections from station A to station B are* feasible *for this query. A feasible connection that is part of a dominant set among all feasible connections is* optimal. *The query's result is a dominant set among all feasible connections.*

Time Query Algorithm. We want to compute the result of a time query $A@\tau \to B$ (Definition 4.5) with a Dijkstra-like algorithm. It stores multiple labels with each station. A *label* represents a connection P stored as a tuple (Z_a, arr)[3], where Z_a is the arrival stop event of P and arr is the arrival time of P. The source station is always A, the target station $S_\text{a}(P)$ is implicitly given by the station that stores this label. Furthermore, we only consider connections departing not earlier than τ at A and want to minimize the arrival time. As we do not further care about the actual departure time at A, we need to define a new dominance relation. To distinguish the new dominance relation (Definition 4.7) from the previous dominance relation (Definition 4.2), we call the considered connections *arrival connections* if we want to use the new dominance relation. All notation and other definitions related to connections therefore apply to arrival connections, too.

[2] Another interesting problem is the latest departure problem (LDP). It is symmetric to the EAP, and symmetric algorithms to the ones that solve the EAP can be used to solve the LDP.

[3] Such a label does not uniquely describe a connection but stores all relevant information for a time query. To compute the actual connection, and not only the earliest arrival time, we need to store an additional parent pointer with each label.

Definition 4.7 *An arrival connection P* dominates *Q iff all of the following conditions are fulfilled:*

(1) $S_a(P) = S_a(Q)$

(2) $\text{arr}(P) \leq \text{arr}(Q)$

(3) $Z_a(P) = Z_a(Q)$*, or Q does not have a critical arrival, or* $\text{ndep}(Q) - \text{arr}(P) \geq \text{transfer}(S_a(P))$

Lemma 4.8 shows that we can compute a dominant set of arrival connections from dominant prefixes.

Lemma 4.8 *Let (A, B, τ) be a time query. A consistent arrival connection P dominates a consistent arrival connection Q iff for all consistent arrival connections R with prefix Q, we can* replace *Q by P to get a consistent arrival connection R' with $S_a(R) = S_a(R')$, and* $\text{arr}(R') \leq \text{arr}(R)$.

Proof. This proof is similar to the one of Lemma 4.4.

$\Rightarrow$: P dominates Q: We need to show that R' is a consistent connection. Condition (1) ensures that $S_a(R) = S_a(R')$. Condition (2) ensures that we can replace $R = Q$ by P directly or when transfer durations are irrelevant. The last condition (3) ensures that we can replace Q by P even when Q is just a prefix of a bigger connection and we need to consider transfer durations. The suffix of this bigger connection w. r. t. Q may depart in $S_a(Q)$ with stop event $Z_a(Q)$ and due to condition (3) it is consistent to transfer from $Z_a(P)$. If $Q \neq R$, then $\text{arr}(R) = \text{arr}(R')$, otherwise, condition (2) ensures that $\text{arr}(R') = \text{arr}(P) \leq \text{arr}(Q) = \text{arr}(R')$.

$\Leftarrow$: $\forall R$ we can replace Q by P: We need to show that all 3 conditions hold to prove that P dominates Q. Condition (1) holds trivially. We get condition (2) with $R = Q$.

Condition (3): Let $Q = (c_1, \ldots, c_k)$ have a critical arrival and $Z_a(Q) \neq Z_a(P)$. Then there is a next departure of the train at station $S_a(Q)$ with elementary connection c_{k+1}. We choose $R := (c_1, \ldots, c_k, c_{k+1})$. As we can replace Q by P in R to get a consistent connection, $\text{ndep}(Q) - \text{arr}(P) \geq \text{transfer}(S_a(P))$ must hold. □

Our algorithm manages a set of dominant **a**rrival **c**onnections $\text{ac}(S)$ for each station S, see Algorithm 4.1 for pseudo-code. The initialization of $\text{ac}(A)$ at the departure station A is a special case since we have no real connection to station A. That is why we introduce a special stop event $\perp$ and we start with the set $\{(\perp, \tau)\}$ at station A. Our query algorithm then knows that we are able to board all trains that depart not earlier than τ. We perform a label correcting query that uses the minimum arrival time of the (new) connections as key of a priority queue. This algorithm needs two elementary operations: (1) *link*: We need to traverse an edge $e = (S, T)$ by linking a given set of arrival connections $\text{ac}(S)$ with the connections $c(e)$ to get a new set of arrival connections to station T. (2) *minima*: We need to combine the already existing arrival connections at T with the new ones to a dominant set. We found a solution to the EAP once we extract a label of station B from the priority queue, as Theorem 4.9 shows.

Algorithm 4.1: TimeQuery(A, B, τ)

input : source station A, target station T, departure time τ
output : earliest arrival time (Definition 4.5)

```
// tentative dominant sets of arr. connections from A to S
1  foreach S ∈ 𝔅 \ A do ac(S) := ∅;
2  ac(A) := {(⊥, τ)};          // special value for A since we depart here at time τ
3  Q.insert(τ, A);             // priority queue, key is earliest arrival time
4  while Q ≠ ∅ do
5      (t, S) := Q.deleteMin();
6      if S = B then return t;                          // done when at target B
7      foreach e := (S, T) ∈ E do
8          N := min(ac(T) ∪ e.link(ac(S)));
9          if N ≠ ac(T) then                            // new connections not dominated
10             ac(T) := N;                              // update arrival connections at T
11             k := min_{P∈N} arr(P);                   // earliest arrival time at T
12             Q.update(k, T);
13 return ⊥;                                            // target B not reachable
```

Theorem 4.9 *Our time query algorithm in the station graph model solves a time query.*

Proof. The query algorithm only creates consistent connections because link and minima do so. Lemma 4.8 ensures that there is never a connection with earlier arrival time. The connections depart from station A not before τ by initialization, so they are feasible for the time query. Since the duration of any connection is non-negative, and by the order in the priority queue, the first label of B extracted from the priority queue has the earliest arrival time at B. □

The link and minima operation dominate the runtime of the query algorithm and we describe efficient algorithms in Section 4.1.5. The most important part is to have a suitable order of the connections, primarily ordered by arrival time. The minima operation is then mainly a linear merge operation, and the link operation uses precomputed intervals to look only at a small relevant subset of $c(e)$. We gain additional speed-up by combining the link and minima operation.

Profile Query Algorithm. We extend the previous algorithm to answer profile queries (Definition 4.6). Our profile query algorithm computes dominant **con**nections $\mathrm{con}(S)$ for each reached station S instead of dominant arrival connections, see Algorithm 4.2. Also we cannot just stop the search when we remove a label of B from the priority queue for the first time. We are only allowed to stop the search when we know that we have a dominant set among *all* feasible connections. For daily operating trains, we can compute a maximum duration for a set of connections and can use it to prune the search. We show the correctness of our profile query algorithm with Theorem 4.10.

Algorithm 4.2: ProfileQuery(A, B)

input : source station A, target station T
output : dominant set among all feasible connections (Definition 4.6)

// tentative dominant sets of connections from A to S
foreach $S \in \mathscr{B} \setminus A$ **do** $\text{con}(S) := \emptyset$;
// special value for A since we depart here
$\text{con}(A) := \{\perp\}$;
// priority queue, key is minimum duration from station A
Q.insert$(0, A)$;
while $Q \neq \emptyset$ **do**
    $(t, S) := Q$.deleteMin();
    // prune due to daily (1440 min. = 1 day) operating trains
    if $\min_{P \in \text{con}(B)} \{\text{arr}(P) - \text{dep}(P)\} + 1440 + \text{transfer}(A) + \text{transfer}(B) \leq t$ **then**
        return $\text{con}(B)$;
    foreach $e := (S, T) \in E$ **do**
        $N := \min(\text{con}(T) \cup e.\text{link}(\text{con}(S)))$;
        if $N \neq \text{con}(T)$ **then** // new connections not dominated
            $c(T) := N$; // update arrival connections at T
            $k := \min_{P \in N} \{\text{arr}(P) - \text{dep}(P)\}$; // min duration from A to T
            Q.update(k, T);
return $\perp$;

Theorem 4.10 *Our profile query algorithm in the station graph model computes a dominant set among all feasible connections.*

Proof. The query algorithm only creates consistent connections because link and minima do so. Furthermore, the minima operation ensures that the stored sets of connections are always dominant. Lemma 4.4 ensures that we can compute for each feasible connection R a feasible connection R' from dominant prefixes, and R' dominates R. Left to prove is showing the correctness of our stopping criterion. We assume daily operating trains. Let P be a connection in $\text{con}(B)$ with minimal $\text{arr}(P) - \text{dep}(P)$. Every connection Q with $\text{arr}(Q) - \text{dep}(Q) \geq \text{arr}(P) - \text{dep}(P) + 1440 + \text{transfer}(A) + \text{transfer}(B)$ can always be dominated by P, or the same connection P' on the next day. □

Efficient algorithms for the link and minima operation are complex, see Section 4.1.5. Similar to a time query, we use a suitable order of the connections, primarily ordered by departure time. The minima operation is an almost linear merge: we merge the connections in descending order and remove dominated ones. This is done with a sweep buffer that keeps all previous dominant connections that are relevant for the current departure time. The link operation, which links connections from station A to S with connections from station S to T, is more complex: in a nutshell, we process the sorted connections from A to S one by one, compute a relevant interval of connections from S

to T as for the time query, and remove dominated connections using a sweep buffer like for the minima operation.

4.1.4 Node Contraction

Here, we show how to perform node contraction (Section 3.2) of a network in the station graph model. After this preprocessing step, we are able to use a faster algorithm to answer time and profile queries.

Contracting a node (= station) v in the station graph removes v and all its adjacent edges from the graph and adds *shortcut edges* to preserve dominant connections between the remaining nodes. A shortcut edge bypasses node v and represents a set of connections. Practically, we contract one node at a time until the graph is empty. All original edges together with the shortcut edges form the result of the preprocessing, a contraction hierarchy (CH).

Preprocessing. The most time consuming part of the contraction is the *witness search*: given a node v and an incoming edge (u,v) and an outgoing edge (v,w), is a shortcut between u and w necessary when we contract v? We answer this question with a one-to-many profile search from u omitting v. We want to find for every connection of the path $\langle u,v,w\rangle$ a dominating connection. In this case, we can omit a shortcut, otherwise we add a shortcut with all the connections that have not been dominated. To keep the number of witness searches small, we maintain a set of necessary shortcuts for each node v. They do not take a lot of space since timetable networks are much smaller than road networks. Then, the contraction of node v is reduced to just adding the stored shortcuts. Initially, we perform a one-to-many witness search from each node u and store with each neighbor v the necessary shortcuts (u,w) that bypass v. The search can be limited by the duration of the longest potential shortcut connection from u.

After the contraction of a node v, we need to update the stored shortcuts of the remaining nodes. The newly added shortcuts (u,w) may induce other shortcuts for the neighbors u and w, see Figure 4.6. So we perform one forward witness search from u and add to w the necessary shortcuts (u,x) bypassing w. A backward witness search from w updates node u. To omit the case that two connections witness each other, we add a shortcut when the witness has the same duration and is not faster. So at most two witness searches from each neighbor of v are necessary. When we add a new shortcut (u,w), but there is already an edge (u,w), we merge both edges using the minima operation, so there are never parallel edges. Avoiding these parallel edges is important for the contraction, which performs worse on dense graphs. Thereby, we also ensure that we can uniquely identify an edge by its endpoints.

We also limit the number of hops and the number of transfers of a witness search [67]. This accelerates the witness search at the cost of potentially more shortcuts.

We do not require loops in static and time-dependent road networks. But for station graph timetable networks, loops are sometimes necessary when transfer durations differ

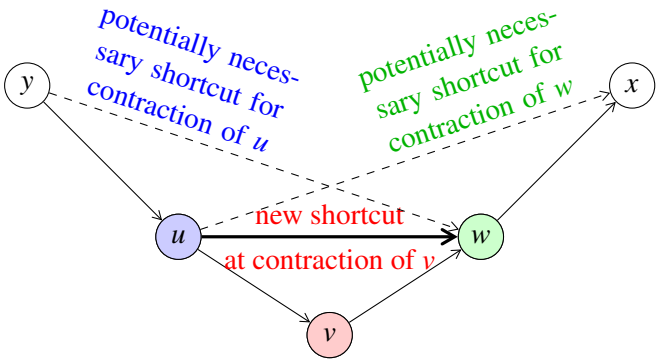


Figure 4.6: If the contraction of node v adds a new shortcut (u,w), we need to update the necessary shortcuts stored with u and w. For the contraction of w (in the future), a potentially necessary shortcut (u,x) needs to be considered and for the contraction of u (in the future), a potentially necessary shortcut (y,w) needs to be considered.

between stations, see Table 4.7. Technically speaking, loops allow to transfer between certain connections at a station below the minimum transfer duration. These loops also make the witness computation and the update of the stored shortcuts more complex. A shortcut (u,w) for node v with loop (v,v) must not only represent the path $\langle u,v,w\rangle$, but also $\langle u,v,v,w\rangle$. So when we add a shortcut (v,v) during the contraction of another node, we need to recompute all stored shortcuts of node v.

Table 4.7: Example showing why loop shortcuts are necessary. The timetable consists of two trains and four stations. Let transfer(B) $= 5$ and transfer(C) $= 1$. Such a timetable is not unrealistic, if for example station B is a large station, whereas station C has just two rail tracks on a single platform. We want to go from station A to station D. It is not consistent to transfer from train 1 to train 2 at station B since it would require a transfer duration of 3 minutes or less. However, it is possible to transfer at station C and then we get a consistent connection from station A to station D arriving at 12:05. Thus, when we contract station C, we need to add a loop at station B.

(a) train 1

station		time
A	dep.	12:00
B	arr.	12:01
	dep.	12:01
C	arr.	12:02

(b) train 2

station		time
C	dep.	12:03
B	arr.	12:04
	dep.	12:04
D	arr.	12:05

The order in which the nodes are contracted is deduced from a node priority consisting of: (a) The edge quotient, the quotient between the number of shortcuts added and the number of edges removed from the remaining graph [136]. (b) The hierar-

chy depth, an upper bound on the amount of hops that can be performed in the resulting hierarchy. Initially, we set $\text{depth}(u) = 0$ and when a node v is contracted, we set $\text{depth}(u) = \max(\text{depth}(u), \text{depth}(v) + 1)$ for all neighbors u. We weight (a) with 10 and (b) with 1 in a linear combination to compute the node priorities. Nodes with higher priority are more "important" and get contracted later. The nodes are contracted in rounds. In each round, nodes with a priority that is minimal in their 2-neighborhood are contracted. We ensure unique priority values by taking the node id into account [136]. Also note that we do not need to perform a simulated contraction of a node to compute its edge quotient [67, 15] due to our stored sets of necessary shortcuts [18].

As mentioned in the introduction, we cannot directly use the algorithms used for time-dependent road networks [15]. We tried using the time-dependent model for the timetable networks, but too many shortcuts were added, especially a lot of shortcuts between the different train-route nodes of the same station pair occur.[4] Additionally, Batz et al. [15] strongly base their algorithm on min-max search that only uses the time-independent min/max duration of an edge to compute upper and lower bounds. However, in timetable networks, the maximum travel time for an edge is very high, for example when there is no service during the night. So the computed upper bounds are too high to bring any exploitable advantages. Without min-max search, the algorithm of Batz et al. [15] is drastically less efficient, as the preprocessing takes days instead of minutes.

Query. The query is based on the ideas of Section 3.2.2, but some augmentations are necessary. Remember that a directed edge (v, w), where w is contracted after v, is an *upward* edge, and where w is contracted before v is a *downward* edge. However, this definition does not consider loops (v, v), a loop is always considered as an upward edge.

For a CH time query, we do not know the arrival time at the target node. We solve this by marking all downward edges that are reachable from the target node. The standard time query algorithm, using only upward edges and the marked downward edges, solves the EAP.

The CH profile query performs a baseline profile query from the source using only upward edges. Additionally it performs a backwards profile query from the target node using only downward edges. The meeting nodes of both search scopes define connections from source to target and the dominant set among all these connections is the result of the CH profile query. The stopping criterion of the search is different, we keep a tentative set of dominant connections from the source station A to the target station B. Every time forward and backward search space meet, we update this tentative set and compute the maximum travel time for arbitrary departure time considering just this set. We can abort the search in a direction once the key of the priority queue is larger than this maximum travel time. Note that using further optimizations that work for road networks, more precisely the stall-on-demand technique and min-max search [15], would even slowdown our query.

[4] We tried to merge train-route nodes but this brought just small improvements.

The restriction of the CH query is that it only computes paths consisting of a sequence of upward edges followed by a sequence of downward edges. Lemma 4.11 proves that due to our preprocessing, we still are able to compute dominant connections. Therefore, our time query (Theorem 4.12) and our profile query (Theorem 4.13) are correct.

Lemma 4.11 *Let Q be a consistent connection. Then there exists a sequence of upward edges $e_1,\dots,e_i$ in the CH and a sequence of downward edges $e_{i+1},\dots,e_j$ in the CH with $P_1 \in c(e_1),\dots,P_j \in c(e_j)$, such that the consistent connection P created from the concatenation of $P_1,\dots,P_j$ dominates Q.*

Proof. WLOG we assume that the stations are numbered $1..n$ by order of contraction, station 1 being contracted first.

Let $X = \langle (e_1,P_1),\dots,(e_k,P_k)\rangle$ be a sequence of edge/connection pairs, we will call it *connection-path*, such that $P_1 \in c(e_1),\dots,P_k \in c(e_k)$, the concatenation of the connections, denoted by X_{con}, is a consistent connection and the concatenation of the edges, denoted by X_{path}, is a path in the graph including the shortcuts. Define $M(X) := \{(a,b) \mid 1 \le a < b \le k$ with $S_{\text{d}}(P_{a+1}) = S_{\text{d}}(P_{a+2}) = \dots = S_{\text{d}}(P_b)$ and $S_{\text{d}}(P_b)$ is contracted before $S_{\text{d}}(P_a), S_{\text{a}}(P_b)\}$. If $M(X) \neq \emptyset$, define $MS(X) := \min\{S_{\text{d}}(P_b) \mid (a,b) \in M(X)\}$. Note that if $M(X) = \emptyset$ the path X_{path} can be split into a sequence of upward edges and a sequence of downward edges as in the description of this lemma.

We will prove that for each connection-path X we can construct a connection-path Y such that Y_{con} dominates X_{con} and $M(Y) = \emptyset$ or $MS(Y) > MS(X)$. This finishes our proof by iteratively applying this construction on the connection-path created from the elementary connections of Q together with the edges that contain these elementary connections. Note that the domination relation on connections is transitive, and that after a finite number of iterations, $M(Y) = \emptyset$ must hold, as there are only a finite number of stations in our graph.

Let $M(X) \neq \emptyset$ and $S_{\text{d}}(P_b) := MS(X)$ with $(a,b) \in M(X)$. The contraction of station $S_{\text{d}}(P_b)$ either finds a witness for the connection $(P_a,\dots,P_b)$ or adds a shortcut with this connection. Let X' be the connection-path of this witness or shortcut for our considered connection. We know that if $M(X') \neq \emptyset$, then $MS(X') > MS(X)$, as we contracted station $MS(X) = S_{\text{d}}(P_b)$. Therefore, we can replace $(e_a,P_a),\dots,(e_b,P_b)$ in X with X'. We do this iteratively for any occurrence of station $MS(X)$ as an endpoint of an edge of X_{path}, as stations can appear several times. For the resulting connection-path Y holds either $M(Y) = \emptyset$ or $MS(Y) > MS(X)$. Also, due to Lemma 4.4, Y_{con} dominates X_{con} by construction. □

Theorem 4.12 *The CH time query solves the EAP.*

Proof. Let Q be a connection solving the EAP for a time query (A,B,τ). With Lemma 4.11 we know that there is a path as a sequence of upward edges followed by a sequence of downward edges describing a connection P that dominates Q. This path can

be found by our CH query algorithm, essentially a baseline time query on a restricted set of edges. Therefore, with Theorem 4.9 we know that we solve the EAP correctly. □

Theorem 4.13 *The CH profile query computes a dominant set among all consistent connections from A to B.*

Proof. Let Q be a consistent connection from A to B. With Lemma 4.11 we know that there is a path as a sequence of upward edges followed by a sequence of downward edges describing a connection P that dominates Q. Therefore, with Theorem 4.10 we know that the forward search computes a connection dominating the subconnection described by the sequence of upward edges, and the backward search computes a connection dominating the subconnection described by the sequence of downward edges. Thus, our CH profile query computes a connection P that dominates Q. As we ordered the priority queue by minimum duration of the connections, we can stop the search as soon as the minimum key in the priority queue exceeds the computed maximum travel time. □

4.1.5 Algorithms for the Link and Minima Operation

The main ideas of our station graph model are simple: Use just one node per station, and no parallel edges. A potential explanation why nobody used it before is that the operations required to run a Dijkstra-like algorithm in this model become quite complex with realistic transfer durations. The following two operations are required:

1. *link*: We need to traverse an edge $e = (S,T)$ by linking a given set of connections from A to S with the connections $c(e)$ to get a new set of connections to station T.

2. *minima*: We need to combine two sets of dominant connections between the same pair of station to a new dominant set among the union of both.

Straightforward implementations of these operations are simple, but their runtime is at least quadratic in the number of connections. We provide implementations that have almost linear runtime. This significantly improves the performance of our query algorithms, and we therefore devote a whole section to these implementations.

The ordered set of connections for an edge is stored in an array. It is primarily ordered by departure time, secondarily by arrival time, and finally critical arrivals come before non-critical arrivals. We assume daily operating connections and give times in minutes. A day has 1 440 minutes, so we store for each connection only one representative with departure time in $[0, 1\,439]$. Therefore, the array actually represents a larger *unrolled* array of multiple days and we get the connections of different days by shifting the times by 1 440. Given two time values t and t', we define the cycledifference(t,t') as smallest non-negative integer ℓ such that $\ell \equiv t' - t \pmod{1\,440}$.

Operations on Arrival Connections

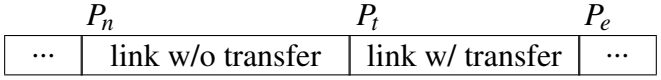
The realistic transfer durations are the main issue why an efficient implementation is complex. Due to them, not only the earliest arriving connection at a station S is dominant, but also ones arriving less than transfer(S) minutes later if they have a *critical arrival*.

We store a set of arrival connections as array. It is primarily ordered by arrival time. This ensures that no arrival connection in the array dominates an arrival connection with lower index.

Consider the relaxation of an edge $e = (S,T)$ during a time query. This requires to link the arrival connections in $\text{ac}(S)$ to the connections in $c(e)$. A basic link algorithm for the time query would pairwise link them together, and afterwards remove the dominated arrival connections. Let $g := |\text{ac}(S)|$ and $h := |c(e)|$. The basic algorithm would create up to $\Theta(g \cdot h)$ arrival connections. Especially h can be very large even though usually only a small range in $c(e)$ is relevant for the link operation.

We improve the basic algorithm. Given a connection $P \in \text{ac}(S)$, we identify the first connection $P_t \in c(e)$ we can transfer to. This first connection is the beginning of a *dominant range*. Obviously, there will be a connection in $c(e)$, so that after this connection, all connections linked with P will result in dominated connections. Therefore, such a connection marks the end of a dominant range. It is preferable to make the dominant range as small as possible, but also supersets of dominant ranges are dominant ranges. We could also distinguish between linking to a certain connection with and without transfer, but we restrict ourselves only to the case with transfer. This results in a practically very efficient link operation since we can precompute the dominant range for each P_t as it is independent of P. Details on how to compute such a dominant range are provided later in this section. So given an array of arrival connections $\text{ac}(S)$ and an array of connections of an edge $c(e)$ to relax, the link will work as follows:

1. $\text{edt} := \min_{P \in \text{ac}(S)} \text{arr}(P) \pmod{1\,440}$ // **e**arliest **d**eparture **t**ime, in $[0, 1\,439]$
2. $\text{ett} := \text{edt} + \text{transfer}(S) \pmod{1\,440}$ // **e**arliest departure with **t**ransfer **t**ime
3. Find first connection $P_n \in c(e)$ with minimal $\text{cycledifference}(\text{edt}, \text{dep}(P_n))$ using buckets.
4. Find first connection $P_t \in c(e)$ with minimal $\text{cycledifference}(\text{ett}, \text{dep}(P_t))$. Connection P_t gives a dominant range that is identified by the first connection P_e outside the range. This partitions the unrolled array of $c(e)$:



We may only link to a connection in $[P_n, P_t)$ without transfers and thus all arrival connections in $\text{ac}(S)$ are relevant to decide which consistent arrival connections we can create there. It is consistent to link to all connections with transfers from P_t on.

5. While we link, we remember the minimal arrival time τ^* and use it to skip dominated arrival connections using Lemma 4.14.
6. Finally, we sort the resulting connections and remove the dominated ones, again using Lemma 4.14. This step is necessary because the minimum arrival time may decrease while we link and we may have to remove duplicates, too.

Lemma 4.14 *Let $\mathscr{S}$ be a set of arrival connections at a station S, and let the mapping $\mathscr{S} \to \mathscr{Z}_S, P \mapsto Z_a(P)$ be injective. Pick $P^* \in \mathscr{S}$ with earliest arrival time $\tau^* := \text{arr}(P^*)$, preferably having a critical arrival. Then $\mathscr{S}^* := \{P^*\} \cup \{P \in \mathscr{S} \mid P \neq P^* \wedge \text{arr}(P) < \tau^* + \text{transfer}(S) \wedge P \text{ has a critical arrival}\}$ is a dominant set among $\mathscr{S}$.*

Proof. We need to show that for each $Q \in \mathscr{S} \setminus \mathscr{S}^*$ there exists a $P \in \mathscr{S}^*$ such that P dominates Q, and no arrival connection $P \in \mathscr{S}^*$ dominates an arrival connection $Q \in \mathscr{S}^* \setminus \{P\}$.

Let $Q \in \mathscr{S} \setminus \mathscr{S}^*$. We will prove that P^* dominates Q, that is all three conditions (1), (2) and (3) of Definition 4.7 are fulfilled. Condition (1) holds as $S_a(P^*) = S = S_a(Q)$. Condition (2) holds as P^* has the earliest arrival time of all connections in $\mathscr{S}$. As $Q \notin \mathscr{S}^*$, either $\text{arr}(Q) \geq \text{arr}(P^*) + \text{transfer}(S)$ or Q does not have a critical arrival. In case that Q does not have a critical arrival, condition (3) holds. Otherwise, $\text{ndep}(Q) \geq \text{arr}(Q) \geq \text{arr}(P^*) + \text{transfer}(S)$, and thus condition (3) holds as well.

Now assume, for the sake of contradiction, that there exists $P \in \mathscr{S}^*$ and $Q \in \mathscr{S}^* \setminus \{P\}$ such that P dominates Q. Condition (3) has to hold, thus either $Z_a(P) = Z_a(Q)$ or Q does not have a critical arrival or $\text{ndep}(Q) - \text{arr}(P) \geq \text{transfer}(S)$. $Z_a(P) = Z_a(Q)$ is not possible as $P \neq Q$ and the mapping $\mathscr{S} \to \mathscr{Z}_S, P' \mapsto Z_a(P')$ is injective. If Q does not have a critical arrival, then $Q = P^*$, as all other arrival connections in $\mathscr{S}^*$ have critical arrivals. As we preferred P^* having a critical arrival, P has to arrive later than $Q = P^*$, as otherwise we would have picked P for P^*. Thus condition (2) cannot hold. If $\text{ndep}(Q) - \text{arr}(P) \geq \text{transfer}(S)$, then $\text{arr}(Q) \geq \text{arr}(P) + \text{transfer}(S) \geq \text{arr}(P^*) + \text{transfer}(S)$. Observe that $\text{transfer}(S) > 0$ has to hold, as otherwise $\mathscr{S}^* = \{P^*\}$. Therefore, $\text{arr}(Q) > \text{arr}(P^*)$ and thus $Q \neq P^*$ and $Q \notin \mathscr{S}^*$, contrary to our assumption. □

Note that the requirement of the mapping $\mathscr{S} \to \mathscr{Z}_S, P \mapsto Z_a(P)$ being injective is no real restriction of Lemma 4.14. If there are multiple connections in $\mathscr{S}$ arriving with the same arrival stop event (same train at same time), we just need to keep one of them.

Example 4.15 *Consider the example of Figure 4.8 of an earliest arrival query from station A with earliest departure time 7:30. Assume that the two marked connections arriving at C currently represent the set of arrival connections* $\text{ac}(C)$*. We want to link* $\text{ac}(C)$ *with* $c(C,D)$ *to compute a set of arrival connections at station D. Note that both* $\text{ac}(C)$ *and* $c(C,D)$ *are already ordered like we would store them in an array. In Step 1 of our link algorithm we compute the earliest departure time* $\text{edt} = 9{:}00$*. Step 2 computes the earliest departure with transfer time* $\text{ett} = 9{:}05$*. Step 3 computes the connection*

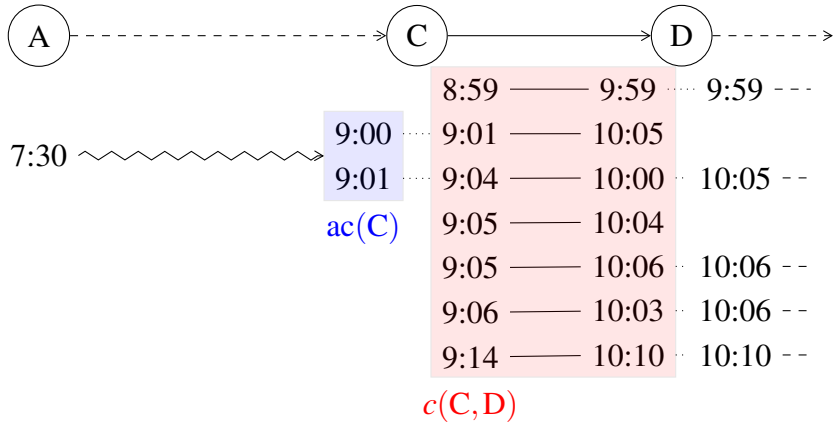


Figure 4.8: Excerpt of an example timetable while performing a time query from station *A* with earliest departure time 7:30. Currently, the time query algorithm settles station C and relaxes edge (C,D). Below the edge (C,D), there is a set of connections, represented by their departure time (left) and their arrival time (right). Riding the train from the left station to the right station is denoted by a solid line. At a station, dotted lines show that there is no transfer necessary. The minimum transfer duration is 5 minutes at each station. Therefore, if the time difference without a transfer at a station is below 5 minutes, it has a critical arrival/departure, for example arriving at D at 9:59 and departing at 9:59 again is a critical arrival/departure, whereas arriving at 10:00 and departing at 10:05 is not.

	scan step	connection P	add	reason	min. arrival time τ^*
w/o transfer	1	9:01 — 10:05	yes	arrival connection 9:00 $\in$ ac(C) can be linked to P w/o transfer, update $\tau^* := 10{:}05$	10:05
	2	9:04 — 10:00	yes	arrival connection 9:01 $\in$ ac(C) can be linked to P w/o transfer, update $\tau^* := 10{:}00$	10:00
w/ transfer	3	9:05 — 10:04	no	τ^* not updated and P does not have a critical arrival	10:00
	4	9:05 — 10:06	no	τ^* not updated and $\text{arr}(P) \geq \tau^* + \text{transfer(D)}$	10:00
	5	9:06 — 10:03	yes	$\text{arr}(P) < \tau^* + \text{transfer(D)}$ and P has a critical arrival	10:00

Table 4.9: Scan over all connections departing not earlier than 9:00. As the minimum transfer duration is 5 minutes, in the range [9:00, 9:04], only linking without transfer is possible. Each connection departing not earlier than 9:05 can be reached with a transfer.

P_n = 9:01 — 10:05, as it is the first connection to depart not earlier than edt $=$ *9:00. Step 4 computes the connection P_t = 9:05 — 10:04, as it is the first connection to depart not earlier than* ett $=$ *9:05. Furthermore, we precomputed another connection P_e for connection P_t that represents a dominant range following Lemma 4.16. Let us look close on how P_e is computed: the departure time of P_t is 9:05, the minimum duration of any connection of $c(C,D)$ is 56 minutes, and the duration of P_t is 59 minutes, therefore d' = 59 - 56 = 3 minutes, and the minimum transfer duration at D is 5 minutes. So, following Lemma 4.16, P_e has to be the first connection that does not depart earlier than 9:05 + 3 + 5 = 9:13, therefore P_e = 9:14 — 10:10. In Step 5 of the link operation, we scan through all the connections starting with P_n and stopping before reaching P_e, see Table 4.9. While we do that, we update the minimum arrival time τ^* and use Lemma 4.14 to prune dominated connections. In Step 6, we remove dominated arrival connections, in this example, only the one added in scan step 1, again using Lemma 4.14. After that, Lemma 4.14 proves that the remaining set of arrival connections is a dominant one.*

The second important operation is the *minima* operation. Given two sets of arrival connections at a node, this operation builds the dominant set among the union. This can be done in linear time by just identifying the minimum arrival time τ^* and using Lemma 4.14. Figure 4.10 shows an example. Sometimes arrival connections are equivalent but not identical. Two arrival connections are *equivalent* if they are identical or

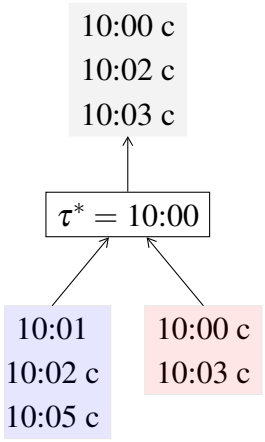


Figure 4.10: Build the minima of two sets of arrival connections arriving at a station S. The two sets are in the bottom of this figure, and for each set the arrival time of each arrival connection is given. An arrival connection with critical arrival is marked with 'c'. The overall minimum arrival time τ^* is 10:00. The minimum transfer duration of station S is 5 minutes. We build a dominant set among the union of the sets using Lemma 4.14. So we first identify our earliest arriving connection, that is the one arriving at $\tau^* :=$ 10:00. Then, we merge both ordered sets of arrival connections. We keep all arrival connections except the one arriving at 10:01 because it does not have a critical arrival, and the one arriving at 10:05 because 10:05 $\geq \tau^* +$ transfer(S).

have the same arrival time and neither of them has a critical arrival. In this case we must keep just one of them. We make this decision so that we minimize the number of queue inserts in the query algorithm, e. g. prefer the one that was not created in the preceding link operation.

Running time. The above minima operation runs in linear time. But the link operation is more complex than a usual link operation that maps departure time to arrival time. However, we give an idea why this link operation is very fast and may work in constant time in many cases. The experiments in Section 4.1.6 show that it is indeed very efficient. Let b be the number of connections in the bucket where P_n is located. Let c_d be the number of connections that depart within the transfer duration window $[P_n, P_t)$ at the station. Let c_a be the number of arrival connections $|\mathrm{ac}(S)|$, r be the number of connections that depart within the range $[P_t, P_e)$. The runtime for computing a link is then $\mathrm{O}(b + c_\mathrm{d} c_\mathrm{a} + r)$. We choose the number of buckets proportional to the number of connections, so b is in many cases constant. For linking connections without transfer, we have the product $\mathrm{O}(c_\mathrm{d} c_\mathrm{a})$ as summand in the runtime. We could improve the product down to $\mathrm{O}(c_\mathrm{d} + c_\mathrm{a} + u)$ with hashing, where u is the number of linked connections. But this is slower in practice, since c_d and c_a are usually very small. The station-dependent minimum transfer duration is usually very small, and also, only very few connections depart and arrive within this transfer duration. It is harder to give a feeling of the size of the range $[P_t, P_e)$. Remember that every connection operates daily. Let be d the difference between the duration of P_t and the minimum duration of any connection in $c(e)$. $d + \mathrm{transfer}(S)$ is an upper bound on the size of the time window of this range. So when d is small, and this should be true in many cases, also r is small. To show that our link operation is empirically fast, we give the average of the parameters we used to bound our runtime in Table 4.11. The average is the most relevant measure because we perform several hundreds link operations per query.

network	b	c_d	c_a	$c_\mathrm{d} \cdot c_\mathrm{a}$	r
PT-EUR	2.7	0.55	1.12	0.68	1.09
PT-VBB	3.0	0.57	1.32	0.94	1.22
PT-RMV	3.2	0.69	1.38	1.20	1.26

Table 4.11: Average of 1 000 random time queries.

Computing the dominant range. Besides the buckets, we also need to compute the dominant ranges. For each connection P_t stored with an edge e, the dominant range that starts with P_t ends with the first connection $P_e \in c(e)$ that does not depart earlier than the time specified by Lemma 4.16. As $c(e)$ is stored as an array that is primarily ordered by departure time, all connections with higher index have no earlier departure time. An efficient algorithm to compute all dominant ranges uses a sweepline algorithm approach.

Lemma 4.16 *Let P be an arrival connection at station S that can link to a connection $Q \in c(S,T)$ with a transfer. Let d' be the difference between the duration of Q and the minimum duration of any connection in $c(S,T)$. Then all connections Q' that depart not earlier than* $\text{dep}(Q) + d' + \text{transfer}(T)$ *will not create a dominant arrival connection when linked with P.*

Proof. Let Q' be a connection that does not depart earlier than $\text{dep}(Q) + d' + \text{transfer}(T)$. By definition of d' is $d(Q) \leq d(Q') + d'$ and thus $\text{arr}(Q) = \text{dep}(Q) + d(Q) \leq \text{dep}(Q) + d(Q') + d' \leq (\text{dep}(Q') - d' - \text{transfer}(T)) + d(Q') + d' \leq \text{arr}(Q') - \text{transfer}(T)$. Thus, the arrival connection created by linking P with Q will always dominate the one created by linking P with Q'. □

Operations on Connections

The operations on connections are more complex than the ones on arrival connections, as we additionally need to consider *critical departures*.

Linking two edges. To link two edges for shortcuts and profile search, we use the dominant range computation at link time. We change the order of the connections in the array for this operation. They are still primarily ordered by departure time. But within the same departure time, the dominant connection should be after the dominated one. That allows for an efficient backward scan to remove dominated connections. We secondarily order by duration descending, thirdly non-critical before critical departure, and then non-critical before critical arrival. Finally, we order by the first and then last stop event, except if the last stop event is critical and the first one not, then we order by the last and then the first stop event. The last criterion is used for an efficient building of a dominant union (minima) of two connection sets where the preference is on one set.

Given two edges $e_1 = (S_1, S_2)$ and $e_2 = (S_2, S_3)$, we want to link all consistent connections to create $c(e_3)$ for a shortcut $e_3 = (S_1, S_3)$. A trivial algorithm would link each consistent pair of connections in $c(e_1)$ and $c(e_2)$ and then compare each of the resulting connections with all other connections to find a dominant set of connections. However, this is impractical for large $g = |c(e_1)|$ and $h = |c(e_2)|$. We would create $\Theta(g \cdot h)$ new connections and do up to $\Theta\big((gh)^2\big)$ comparisons.

So we propose a different strategy for linking that is considerably faster for practical instances. We process the connections in $c(e_1)$ in descending order. Given a connection $P \in c(e_1)$, we want to find a connection $Q \in c(e_1)$ that dominates P at the departure at S_1 (first half of condition (2), and condition (3) of Definition 4.2). So we only need to link P to connections in $c(e_2)$ that depart in S_2 before Q could consistently transfer to them, see Figure 4.12. Preferably we want to find the Q with the earliest arrival time. However, we find the Q with the earliest arrival time at S_2 with $\text{dep}(Q) \geq \text{dep}(P) + \text{transfer}(S_1)$. Then Q will not only dominate P at the departure but also any connection departing not later than P. So we can use a simple finger search to find Q in $c(e_1)$. We also use finger

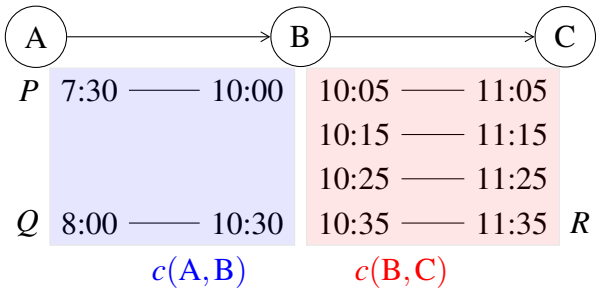


Figure 4.12: Excerpt of a timetable. Each edge has a set of connections, represented by their departure time (left) and their arrival time (right). The minimum transfer duration is 5 minutes at each station. We only need to consider linking P to the first three connections in $c(\mathrm{B},\mathrm{C})$ (of which only the first one will result in a dominant connection). The connection created by linking P with the forth connection R is dominated by the one created by linking Q with R.

search to find the first connection that departs in $c(e_2)$ after the arrival of P. Of course, we need to take the transfer duration at S_2 into account when we link. It is not always necessary to link to all connections that depart in the dominant range specified by the arrival time of P and Q; we can use the knowledge of the minimum duration in $c(e_2)$ to stop linking when we cannot expect any new dominant connections. The newly linked connections may not be dominant and also may not be in order and we fix this by these three measures:

1. We use a sweep buffer that has as state the current departure time and holds all relevant connections with higher order to dominate a connection with the current departure time. The relevant connections are described by the set $\mathscr{S}^*$ in Lemma 4.17. The number of them is usually small, see Figure 4.13. When we link a new connection, we drop it in case that it is dominated by a connection in the sweep buffer. If it is not dominated, we add it to the sweep buffer.

   When the current departure time changes, we update the sweep buffer and drop all connections that do not fulfill the conditions of Lemma 4.17 anymore. Note that only the new departure time, and the time τ' defined in Lemma 4.17 are necessary to update the sweep buffer. Assuming that only few connections depart in S_1 within transfer(S_1) minutes, and only few connections arrive in S_3 within transfer(S_3) minutes, the sweep buffer has only few entries.

2. Connections can only be unordered within a range with the same departure time, for example when they have ascending durations. As there are usually only few connections with same departure time, it is efficient to use the idea of insertion sort to reposition a connection that is not in order. While we reposition a new connection, we must check whether it dominates the connections that are now

positioned before it. For example, a new connection with same departure than the previous one but smaller duration may dominate the previous one if the departure is not critical.

3. After we processed all connections in $c(e_1)$, we have a superset of $c(e_3)$ that is already ordered, but some connections may still be dominated. This happens when the dominant connection departs after midnight and the dominated connection before, so the periodic border is between them. To remove all dominated connections, we continue scanning backwards through the new connections but now on "day -1" using the sweep buffer. We can stop when no connection in the sweep buffer is of "day 0".

Lemma 4.17 *Let $\mathscr{S}$ be a set of connections between stations S and T, and let Q be a connection with $S_{\mathrm{d}}(Q) = S$ and $S_{\mathrm{a}}(Q) = T$. Furthermore, for all $P \in \mathscr{S}$ holds*

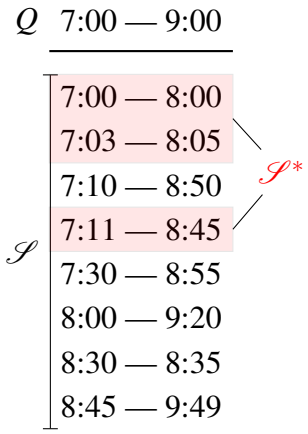


Figure 4.13: Example on how the sweep buffer works. We show connections from a station S to a station T represented by their departure time (left) and arrival time (right). All connections here have distinct stop events, and all have critical arrival and critical departure. The minimum transfer duration at stations S and T is 5 minutes. To decide whether connection Q is dominated by the set of connections $\mathscr{S}$, it is sufficient to look only at the small number of marked connections in $\mathscr{S}^*$. The time τ' from Lemma 4.17 is $\tau' = 8{:}45$. Therefore, our sweep buffer only contains connections departing earlier than $\mathrm{dep}(Q) + \mathrm{transfer}(S) = 7{:}05$, and connections arriving earlier than $\tau' + \mathrm{transfer}(T) = 8{:}50$. In our example, connection Q is dominated by connection $P' := 7{:}10 — 8{:}50 \in \mathscr{S} \setminus \mathscr{S}^*$, because there is at least $\mathrm{transfer}(S) = 5$ minutes time between the departure of Q and the departure of P', and at least $\mathrm{transfer}(T) = 5$ minutes time between the arrival of P' and the arrival of Q. The connection P'' from the proof of Lemma 4.17 is $P'' := 7{:}11—8{:}45 \in \mathscr{S}^*$. And clearly, P'' also dominates Q. But note that P'' does not dominate P', as P' has a critical arrival.

$\text{dep}(Q) \leq \text{dep}(P)$. *Define* $\tau' := \min\{\text{arr}(P) \mid P \in \mathscr{S} \wedge \text{dep}(P) \geq \text{dep}(Q) + \text{transfer}(S)\}$. *Iff there exists* $P' \in \mathscr{S}$ *such that* P' *dominates* Q*, then there exists* $P'' \in \mathscr{S}^* := \{P \in \mathscr{S} \mid \text{dep}(P) < \text{dep}(Q) + \text{transfer}(S)\} \cup \{P \in \mathscr{S} \mid \text{arr}(P) < \tau' + \text{transfer}(T)\}$ *such that* P'' *dominates* Q.

Proof. We have to prove that if exists $P' \in \mathscr{S} \setminus \mathscr{S}^*$ such that P' dominates Q, then there exists $P'' \in \mathscr{S}^*$ that dominates Q. Pick $P'' \in \{P \in \mathscr{S} \mid \text{dep}(P) \geq \text{dep}(Q) + \text{transfer}(S)\}$ with $\text{arr}(P'') = \tau'$. We will prove that P'' dominates Q, that is, all four conditions (1), (2), (3) and (4) of Definition 4.2 hold. Condition (1) holds as $S_\text{d}(P'') = S = S_\text{d}(Q)$ and $S_\text{a}(P'') = T = S_\text{a}(Q)$. Condition (2): $\text{dep}(Q) \leq \text{dep}(P'')$ holds by prerequisite. As $P' \notin \{P \in \mathscr{S} \mid \text{arr}(P) < \tau' + \text{transfer}(T)\}$, we know that $\text{arr}(P') \geq \tau' + \text{transfer}(T)$. And because P' dominates Q, by condition (2), we know that $\text{arr}(Q) \geq \text{arr}(P')$. Both inequalities together prove that $\text{arr}(Q) \geq \tau' + \text{transfer}(T) \geq \tau' = \text{arr}(P'')$, therefore, condition (2) holds as well. Condition (3): Assume that $Z_\text{d}(P'') \neq Z_\text{d}(Q)$ and Q has a critical departure, as otherwise, condition (3) holds directly. By choice of P'' we know that $\text{dep}(P'') \geq \text{dep}(Q) + \text{transfer}(S)$, and therefore, $\text{dep}(P'') - \text{parr}(Q) \geq \text{dep}(P'') - \text{dep}(Q) \geq \text{transfer}(S) \Rightarrow$ condition (3) holds. Condition (4): Assume that $Z_\text{a}(P'') \neq Z_\text{a}(Q)$ and Q has a critical arrival, as otherwise, condition (4) holds directly. As P' dominates Q, by condition (2), $\text{arr}(P') \leq \text{arr}(Q)$ holds, and also $\text{arr}(P') \geq \text{arr}(P'') + \text{transfer}(T)$ because $P' \notin \mathscr{S}^*$. Therefore, $\text{ndep}(Q) - \text{arr}(P'') \geq \text{arr}(Q) - \text{arr}(P'') \geq \text{arr}(Q) - \text{arr}(P') + \text{transfer}(T) \geq 0 + \text{transfer}(T) \Rightarrow$ condition (4) holds. □

Note that in comparison to the link operation for arrival connections, where we could use the minimum arrival time τ^* of Lemma 4.14 to find dominated arrival connections, for connections, the sweep buffer $\mathscr{S}^*$ is the suitable replacement for τ^*.

Running time. We give an idea why this link operation is very fast and may work in linear time in many cases. The experiments in Section 4.1.6 show that it is indeed very efficient. Let c_P be the size of the range in $c(e_2)$ that departs between the arrival of $P \in c(e_1)$ and $Q \in c(e_1)$, as in the description of our algorithm. Let b_P be the runtime of the finger search to find the earliest connection in $c(e_2)$ that departs after the arrival of P. Let s be the maximum number of connections in the sweep buffer. The runtime of link is then $\text{O}\big(\sum_{P \in c(e_1)} (c_P \cdot s + b_P)\big)$. This upper bound reflects the linking and usage of the sweep buffer. The ordering is also included, as s is an upper bound on the number of connections with same departure time, as all these connections are in the sweep buffer. The backward scanning on "day -1" is also included, since it just adds a constant factor to the runtime. The finger search for Q in $c(e_1)$ is amortized in $\text{O}(1)$, so it is also included in the runtime above. It is hard to get a feeling for c_P and b_P, they can be large when $h = |c(e_2)|$ is much larger than $g = |c(e_1)|$. Under the practical assumption that $\sum_{P \in c(e_1)} (c_P + b_P) = \text{O}(g + h)$, we get a runtime of $\text{O}((g+h)s)$. As we already argued when we described the sweep buffer, s is small and in many cases constant, so our runtime should be $\text{O}(g+h)$ in many cases. We give the average of the parameters

we used to bound our runtime in Table 4.14 to show that our link and minima operation are fast in practice.

network	c_P	$\frac{\sum_{P \in c(e_1)} c_P}{g+h}$	b_P	$\frac{\sum_{P \in c(e_1)} b_P}{g+h}$	$\frac{\sum_{P \in c(e_1)} (c_P + b_P)}{g+h}$	s
PT-EUR	0.93	0.29	3.25	0.84	1.14	2.34
PT-VBB	1.26	0.29	3.97	0.93	1.22	3.85
PT-RMV	1.44	0.30	4.30	1.08	1.38	5.30

Table 4.14: Average of 1 000 random profile queries.

Constructing the Minima of two Sets of Connections. Our algorithm for the minima operation is a backwards merge of the ordered arrays of connections that uses the sweep buffer defined by Lemma 4.17, as for the link operation. We also continue backward scanning on "day -1" to get rid of dominated connections over the periodic border. The minima operation is not only used during a query, but also to compare witness paths and possible shortcuts, and to merge parallel edges.

Similar to equivalent arrival connections, two connections are *equivalent* when they have the same duration, an equivalent departure and equivalent arrival. Two connections P and Q have an *equivalent departure* when their departure is identical or when the departure is not critical and they have the same departure time. Analogously, two connections P and Q have an *equivalent arrival* when their arrival is identical or when the arrival is not critical and they have the same arrival time. Note that equivalent connections are stored next to each other in an ordered array. So we can easily detect them during the merge and keep just one of them. To reduce the number of priority queue operations during a search, we prefer a connection that was not created in the preceding link operation.

Running time. Let g and h be the cardinalities of the two sets we merge. Let s be the maximum size of the sweep buffer. Then, the runtime of the minima operation is $O((g+h)s)$. Since s is small and in many cases constant, the runtime should be $O(g+h)$ in many cases.

Integrating Link and Minima. A minima operation always follows a link operation when we relax an edge to an already reached station T. This happens quite often for profile queries, so we can exploit this to tune our algorithm. We directly merge the newly linked connections one by one with the current connections at T. Our sweep buffer can contain current connections and newly linked connections at the same time. When a new connection is not in order, we fix this with the insertion sort idea. This integration reduces required memory allocations and gives significant speed-ups.

4.1.6 Experiments

Environment. The experimental evaluation was done on one core of a Intel Xeon X5550 processor (Quad-Core) clocked at 2.67 GHz with 48 GiB of RAM[5] and 8 MiB of Cache running SuSE Linux 11.1 (kernel 2.6.27). The program was compiled by the GNU C++ compiler 4.3.2 using optimization level 3.

Instances. We have used real-world data from the European railways. The network of the long distance connections of Europe (PT-EUR) is from the winter period 1996/97. The network of the local traffic in Berlin/Brandenburg (PT-VBB) and of the Rhein/Main region in Germany (PT-RMV) are from the winter period 2000/01. The sizes of all networks are listed in Table 4.15.

Table 4.15: Network sizes and number of nodes and edges in the graph for each model.

		trains/	elementary	time-dependent		station based	
network	stations	buses	connections	nodes	edges	nodes	edges
PT-EUR	30 517	167 299	1 669 666	550 975	1 488 978	30 517	88 091
PT-VBB	12 069	33 227	680 176	228 874	599 406	12 069	33 473
PT-RMV	9 902	60 889	1 128 465	167 213	464 472	9 902	26 678

Results. We selected 1 000 random queries and give average performance measures. We compare the time-dependent model and our new station model using a simple unidirectional Dijkstra algorithm in Table 4.16. Time queries have a good query time speed-up above 4.5 and even more when compared to the number of delete mins. However, since we do more work per delete min, this difference is expected. Profile queries have very good speed-up around 5 to 8 for all tested instances. Interestingly, our speed-up of the

[5]We never used more than 556 MiB of RAM, reported by the kernel.

Table 4.16: Performance of the station graph model compared to the time-dependent model on plain Dijkstra queries. We report the total *space*, the *#delete mins* from the priority queue, query *times*, and the *speed-up* compared to the time-dependent model.

			TIME-QUERIES				PROFILE-QUERIES			
		space	#delete	spd	time	spd	#delete	spd	time	spd
network	model	[MiB]	mins	up	[ms]	up	mins	up	[ms]	up
PT-EUR	time-dep.	27.9	259 506	1.0	54.3	1.0	1 949 940	1.0	1 994	1.0
	station	48.3	14 504	17.9	9.4	5.8	48 216	40.4	242	8.2
PT-VBB	time-dep.	11.3	112 683	1.0	20.9	1.0	1 167 630	1.0	1 263	1.0
	station	19.6	5 969	18.9	4.0	5.2	33 592	34.8	215	5.9
PT-RMV	time-dep.	10.9	87 379	1.0	16.1	1.0	976 679	1.0	1 243	1.0
	station	29.3	5 091	17.2	3.5	4.6	27 675	35.3	258	4.8

number of delete mins is even better than for time queries. We assume that more re-visits occur since there are often parallel edges between a pair of stations represented by its train-route nodes. Our model does not have this problem since we have no parallel edges and each station is represented by just one node. It is not possible to compare the space consumption per node since the number of nodes is in the different models different. So we give the absolute memory footprint: it is so small that we did not even try to reduce it, although there is some potential.

Before we present our results for CH, we would like to mention that we were unable to contract the same networks in the time-dependent model. The contraction took days and the average degree in the remaining graph exploded. Even when we contracted whole stations with all of its train-route nodes at once, it did not work. It failed since the necessary shortcuts between all the train-route nodes multiplied quickly, see Example 4.1. So we developed the station graph model to fix these problems. Table 4.17 shows the resulting preprocessing and query performance. We get preprocessing times between 3 to 4 minutes using a hop limit of 7. The number of transfers is limited to the maximal number of transfers of a potential shortcut + 2. These timings are exceptional low (minutes instead of hours) compared to previous publications [39, 25] and reduce time queries below 550 μs for all tested instances. CH work very well for PT-EUR where we get speed-ups of more than 37 for time queries and 65 for profile queries. Compared to plain Dijkstra queries using the time-dependent model (Table 4.16), we even get a speed-up of 218 (time) and 534 (profile) respectively. These speed-ups are one order of magnitude larger than previous speed-ups [39]. The network PT-RMV is also suited for CH, the ratio between elementary connections and stations is however very high, so there is more work per settled node. More difficult is PT-VBB; in our opinion, this network is less hierarchically structured. We see that on the effect of different hop limits for precomputation. (We chose 7 as a hop limit for fast preprocessing and then selected 15 to show further trade-off between preprocessing and query time.) The smaller hop limit increases time query times by about 25%, whereas the other two networks just suffer an

Table 4.17: Performance of CH. We report the preprocessing *time*, the *space* overhead and the increase in edge count. For query performance, we report the *#delete mins* from the priority queue, query *times*, and the *speed-up* over a plain Dijkstra (Table 4.16).

		PREPROCESSING			TIME-QUERIES				PROFILE-QUERIES			
network	hop-limit	time [s]	space [MiB]	edge inc.	#del. mins	spd up	time [μs]	spd up	#del. mins	spd up	time [ms]	spd up
PT-EUR	7	210	45.7	88%	192	75.7	251	37.5	260	186	3.7	65.1
	15	619	45.3	86%	183	79.3	216	43.5	251	192	3.4	71.4
PT-VBB	7	216	27.9	135%	207	28.8	544	7.3	441	76	27.0	8.0
	15	685	26.9	128%	186	32.1	434	9.2	426	79	24.2	8.9
PT-RMV	7	167	36.0	123%	154	33.1	249	14.0	237	117	9.5	27.1
	15	459	35.0	117%	147	34.6	217	16.1	228	121	8.2	31.3

increase of about 16%. So important witnesses in PT-VBB contain more edges, indicating a lack of hierarchy.

We do not really have to worry about preprocessing space since those networks are very small. The number of edges roughly doubles for all instances. We observe similar results for static road networks [67], but there we can save space with bidirectional edges. In timetable networks, we do not have bidirectional edges with the same weight, so we need to store them separately. CH on timetable networks are inherently space efficient as they are event-based, they increase the memory consumption by not more than a factor 2.4 (PT-VBB: 19.6 MiB $\rightarrow$ 47.5 MiB). In contrast, time-dependent road networks are not event-based and can get very complex travel time functions on shortcuts, leading to an increased memory consumption (Germany midweek: 0.4 GiB $\rightarrow$ 4.4 GiB [15]). Only with sophisticated approximations, it is possible to reduce space consumption while answering queries exactly [16].

4.2 Fully Realistic Routing

4.2.1 Central Ideas

In the previous section we investigated how to use node contraction in the restricted scenario with realistic transfer durations. However, in a fully realistic scenario we need new techniques, especially for routing in poorly structured public transportation networks that have only little hierarchy. We present a new algorithm for routing on public transportation networks that is fast even when the network is realistically modeled, very large and poorly structured. These are the challenges faced by public transportation routing on Google Maps (`http://www.google.com/transit`), and our algorithm has successfully addressed them. It is based on the following new idea.

Think of the query $A@\tau \to B$, with source station $A =$ Freiburg, target station $B =$ Zürich, and departure time $\tau =$ 10:00. Without assuming anything about the nature of the network and without any precomputation, we would have to do a Dijkstra-like search and explore many nodes to compute an optimal connection. Now let us assume that we are given the following additional information: each and every optimal connection from Freiburg to Zürich, no matter on which day and at which time of the day, either is a direct connection (with no transfer in between) or it is a connection with exactly one transfer at Basel. We call *Freiburg – Zürich* and *Freiburg – Basel – Zürich* the optimal *transfer patterns* between Freiburg and Zürich (for each optimal connection, take the source station, the sequence of transfers, and the target station). Note how little information the set of optimal transfer patterns for this station pair is. Additionally, let us assume that we have timetable information for each station that allows us to very quickly determine the next *direct* connection from a given station to some other station.

With this information, it becomes very easy to answer the query $A@\tau \to B$ for an arbitrary given time τ. Say $\tau =$ 10:00. Find the next direct connection from Freiburg to Zürich after τ. Say it leaves Freiburg at 12:55 and arrives in Zürich at 14:52. (There are only few direct trains between these two stations over the day.) Also find the next direct connection from Freiburg to Basel after τ. Say it leaves Freiburg at 10:02 and arrives in Basel at 10:47. Then find the next direct connection from Basel to Zürich after 10:47. Say it leaves Basel at 11:07 and arrives in Zürich at 12:00. In our cost model (see Section 2.3.2) these two connections are incomparable (one is faster, and the other has less transfers), and thus we would report both. Since the two given transfer patterns were the only optimal ones, we can be sure to have found all optimal connections. And we needed only three direct-connection queries to compute them.

Conceptually, our whole scheme goes as follows. The set of all optimal transfer patterns between all station pairs is too large to precompute and store. We therefore precompute a set of *parts of* transfer patterns such that all optimal transfer patterns can be combined from these parts. For our three datasets, we can precompute such parts in 20–40 core hours per 1 million departure/arrival events and store them in 10–50 MiB per 1000 stations. From these parts, also non-optimal transfer patterns can be combined,

but this only means additional work at query time; it will not let us miss any optimal connections. Think of storing parts of transfer patterns, to be recombined at query time, as a lossy compression of the set of all optimal transfer patterns. We also precompute a data structure for fast direct-connection queries, which, for our three datasets, needs 3–10 MiB per 1 000 stations and has a query time of 2–10 μs.

Having this information precomputed, we then proceed as follows for a given query $A@\tau \to B$. From the precomputed parts, we compute all combinations that yield a transfer pattern between A and B. We overlay all these patterns to form what we call the *query graph*. Finding the optimal connection(s) amounts to a shortest-path computation on the query graph with source A and target B, where each edge evaluation is a direct-connection query. The query graph for our simple example from above has three nodes (A = Freiburg, B = Zürich, and C = Basel) and three edges (A $\to$ B, A $\to$ C, C $\to$ B). Due to the non-optimal transfer patterns that come from the recombination of the precomputed parts, our actual query graphs typically have several hundreds of edges. However, since direct-connection queries can be computed in about 10 μs, this will still give us query times on the order of a few milliseconds, and by the way our approach works, these times are independent of the size of the network.

4.2.2 Query

In the fully realistic scenario, we only consider queries with a specific departure time. This is similar to a time query (Definition 4.5 in Section 4.1.3) but is based on a multi-criteria cost model. A station-to-station query computes optimal connections between a pair of stations (Definition 4.18). The more realistic scenario of location-to-location queries is introduced in Section 4.2.6 after we extend the definition of a consistent connection in Section 4.2.5 to allow walking.

Definition 4.18 *A* station-to-station query *is given as $A@\tau \to B$, where A is the source station, τ is the earliest departure time, and B is the target station. All consistent connections from station A to station B that do not depart before τ are* feasible *for this query, but their cost is increased by the waiting time between τ and the departure of the connection at A. If the cost of a feasible connection is not dominated by any other, we call them* optimal cost *and* optimal connection*, respectively, for the query.*

The query's result is an optimal set of connections*, that is, a set of optimal connections containing exactly one for each optimal cost.*

We define the desired results for both types of query in the time-expanded graph extended by a source and a target node. All times are represented in seconds since midnight of the timetable's day 0. We can compute the result of a station-to-station query in the time-expanded graph using Corollary 4.19.

Corollary 4.19 *Consider a* station-to-station query $A@\tau \to B$. *Take the first transfer node At$@\tau'$ with $\tau' \geq \tau$. For this query, we extend the time-expanded graph by a source*

node S with an edge of duration $\tau' - \tau$ and penalty 0 that leads to $A\text{t}@\tau'$ and by a target node T with incoming edges of zero cost from all arrival nodes of B.

Exactly the paths from S to T are the feasible connections for the query. Each of these paths has the cost of the feasible connection it represents.

Note that the waiting chain at A makes paths from S through *all* departure nodes after time τ feasible. We exclude multiple connections for the same optimal cost. They do occur (even for a single-criterion cost function) but add little value.[6]

4.2.3 Basic Algorithm

In this section we present a first simple algorithm that illustrates the key ideas. It has very fast query times but a quadratic precomputation complexity.

Fast direct-connection queries

Definition 4.20 *For a* direct-connection query $A@\tau \to B$*, the feasible connections are defined as in Definition 4.18, except that only* connections without transfers *are permitted. The result of the query are the optimal costs in this restricted set.*

The following data structure answers direct-connection queries in about $10\,\mu$s.

1. Group all trains of the timetable into *lines* L1, L2, ... such that all trains on a line share the same sequence of stations, do not overtake each other (FIFO property, like *train routes* [121]), and have the same penalty score between any two stations.

   The trains of a line are sorted by time and stored in a 2D array like this:

line L17	S154	S097		S987		S111		...
train 1	8:15	8:22	8:23	8:27	8:29	8:38	8:39	...
train 2	9:14	9:21	9:22	9:28	9:28	9:37	9:38	...
...	...	...	...	...	...	...	...	...

2. For each station, precompute the sorted list of lines incident to it and its position(s) on each line. For example:

   S097: (L8, 4) (L17, 2) (L34, 5) (L87, 17) ...
   S111: (L9, 1) (L13, 5) (L17, 4) (L55, 16) ...

3. To answer a direct-connection query $A@\tau \to B$, intersect the incidence lists of A and B. For each occurrence of A before B on a line, read off the cost of the earliest feasible train, then choose the optimal costs among all these.

[6] Connections of equal cost, relative to query time τ, arrive at the same time. It is preferable to choose one that departs as late as possible from A; we will return to that in Section 4.2.8. Those that depart latest often differ in trivial ways (e. g., using this or that tram between two train stations), so returning just one is fine.

In our example, the query S097@9:03 $\rightarrow$ S111 finds positions 2 and 4 on L17 and the train that reaches S111 at 9:37.

Lemma 4.21 *A query $A@\tau \rightarrow B$ to the direct-connection data structure returns all optimal costs of direct connections.*

Proof. Each direct connection uses at most one train, as no transfers are allowed. The non-overtaking and same-penalty constraints on the formation of lines implies that, for each occurrence, feasible trains after the first on the same line do not achieve a better cost. Hence Step 3 finds all optimal costs. □

Transfer patterns precomputation

Definition 4.22 *For any connection, consider the stations where a transfer happens. The sequence of these stations is the* transfer pattern *of the path.*

An optimal set of transfer patterns *for a pair (A,B) of stations is a set $\mathscr{T}$ of transfer patterns such that for all queries $A@\tau \rightarrow B$ there is an optimal set of connections whose transfer patterns are contained in $\mathscr{T}$, and such that each element in $\mathscr{T}$ is the transfer pattern of an optimal connection for a query $A@\tau \rightarrow B$ at some time τ.*

For every source station A, we compute optimal sets of transfer patterns to all stations B reachable from it and store them in one DAG for A. This DAG has three different types of nodes: one *root node* labeled A, for each reachable station B a *target node* labeled B, and for each transfer pattern prefix $AC_1 \ldots C_i$, occurring in at least one transfer pattern $AC_1 \ldots C_i \ldots B$, a *prefix node* labeled C_i. They are connected in the natural way such that precisely the transfer patterns we want to store are represented by a path from their target stations to the root node, labeled in reverse order; Figure 4.18 shows an example.

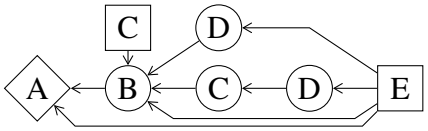


Figure 4.18: DAG for the transfer patterns 'AE', 'ABE', 'ABC', 'ABDE' and 'ABCDE'. The root node is the diamond, prefix nodes are circles and target nodes are rectangles. There are potentially several prefix nodes with the same label: In our example, 'D' occurs twice, the top one representing the prefix 'ABD' and the bottom one 'ABCD'.

We use the following algorithm *TransferPatterns(A)*.

1. Run a multi-criteria variant of Dijkstra's algorithm [105, 134, 100] starting from labels of cost zero at all *transfer* nodes of station A.

2. For every station B, choose optimal connections with the *arrival chain algorithm*: For all distinct arrival times $\tau_1 < \tau_2 < \ldots$ at B, select a dominant subset in the set of labels consisting of (i) those settled at the arrival node(s) at time τ_i and (ii) those selected for time τ_{i-1}, with duration increased by $\tau_i - \tau_{i-1}$; ties to be broken in preference of (ii).

3. Trace back the paths of all labels selected in Step 2. Create the DAG of transfer patterns of these paths by traversing them from the source A.

Lemma 4.23 *If c is an optimal cost for the station-to-station query $A@\tau_0 \to B$,* TransferPatterns(A) *computes the transfer pattern of a feasible connection for the query that realizes cost c.*

Proof. The optimal cost c is the cost of an optimal path $S \to P_1 \to T$ in the extended time-expanded graph from Corollary 4.19, where P_1 starts at some node $A\mathrm{t}@\tau_\mathrm{d}$ (the successor of S) and ends at some node $B\mathrm{a}@\tau_\mathrm{a}$.

We will now attach alternative source and target nodes S' and T' to the graph that reflect the transfer patterns computation. S' has an edge of duration 0 and penalty 0 to all transfer nodes of A. This reflects the initial labels. All arrival nodes $B\mathrm{a}@\tau$ with $\tau \le \tau_\mathrm{a}$ have an edge of duration $\tau_\mathrm{a} - \tau$ and penalty 0 to T'. Hence T' corresponds to the label set for arrival time τ_a. A transfer pattern is computed for a path P_2' such that $S' \to P_2' \to T'$ has better or equal cost than $S' \to P_1 \to T'$. In particular, P_2' departs from A no earlier than P_1 does.[7] That means, we can prepend to P_2' the part of the waiting chain of A between the first node of P_1 and the first node of P_2'; let P_2 denote this extension of P_2'.

To prove that P_2 is the claimed path, it remains to show that $S \to P_1 \to T$ and $S \to P_2 \to T$ have the same cost. By construction of S', T' and by choice of P_2', the following inequalities hold for duration and penalty of the paths:

$$d(S \to P_1 \to T) = \tau_\mathrm{a} - \tau_0 = d(S \to P_2 \to T') \ge d(S \to P_2 \to T),$$
$$p(S \to P_1 \to T) = p(S' \to P_1 \to T') \ge p(S' \to P_2' \to T') = p(S \to P_2 \to T).$$

By optimality of $S \to P_1 \to T$, equality holds throughout. □

Running *TransferPatterns(A)* for all stations A is easy to parallelize by splitting the set of source stations A between machines, but even so, the total running time remains an issue. We can estimate it as follows. Let s be the number of stations, let n be the average number of nodes per station (< 569 for all our graphs, with the optimizations of Section 4.2.8), and let ℓ be the average number of settled labels per node and per run of *TransferPatterns(A)* (< 16 in all our experiments, in the setting of Sections 4.2.8 and 4.2.9). Then the total number of labels settled by *TransferPatterns(A)* for all stations A is $L = \ell \cdot n \cdot s^2$.

[7] We see here that transfer patterns are computed for paths that are optimal in the sense of Definition 4.18 *and* depart as late as possible, see Footnote 6 on page 82.

Steps 1–3 have running time essentially linear in L, with logarithmic factors for maintaining various sets. (For Step 1, this includes the priority queue of Dijkstra's algorithm. The bounded out-degree of our graphs bounds the number of unsettled labels linearly in L.) Since L is quadratic in s, this precomputation is infeasible in practice for large networks, despite parallelization. We will address this issue in Sections 4.2.4, 4.2.7 and 4.2.9.

Query graph construction and evaluation

For a query $A@\tau \to B$, we build the *query graph* as follows, independent of τ:

1. Fetch the precomputed transfer patterns DAG for station A.
2. Search target node B in the DAG. Assume it has ℓ successor nodes with labels $C_1,\ldots,C_\ell$. Add the edges $(C_1,B),\ldots,(C_\ell,B)$ to the query graph.
3. Recursively perform Step 2 for each successor node with a label $C_i \neq A$.

Figure 4.19 shows the query graph from A to E built from the DAG in Figure 4.18.

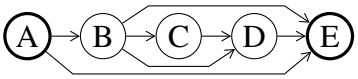


Figure 4.19: Query graph A→ E from transfer patterns 'AE', 'ABE', 'ABDE' and 'ABCDE'.

Lemma 4.24 *For each transfer pattern $AC_1\ldots C_kB$ in the DAG there exists the path $\langle A,C_1,\ldots,C_k,B\rangle$ in the constructed query graph.*

Proof. Let $AC_1\ldots C_kB$ be a transfer pattern in the DAG. Induction over i shows that there is a path $\langle C_i,\ldots,C_k,B\rangle$ in the query graph, and the node C_i in the query graph gets processed together with the prefix node '$AC_1\ldots C_{i-1}$' in the DAG (Steps 2 and 3). Finally, C_1 is processed with the root node 'A' in the graph, so that also the edge (A,C_1) is added and the path $\langle A,C_1,\ldots,C_k,B\rangle$ exists in the query graph. □

Given the query graph, evaluating the query is simply a matter of a time-dependent multi-criteria Dijkstra search [52] on that graph. Labels in the queue are tuples of station and cost (time and penalty). Relaxing an edge $C \to D$ for a label with time τ amounts to a direct-connection query $C@\tau \to D$.

By storing parent pointers from each label to its predecessor on the shortest path, we eventually obtain, for an optimal label at the target station, the sequence of transfers on an optimal path from the source to the target, as well as the times at which we arrive at each of these transfers. More details on the optimal paths can be provided by augmenting the direct-connection data structure.

Theorem 4.25 *For a given query $A@\tau \to B$, the described search on the query graph from A to B returns the set of optimal costs and for each such cost a corresponding path.*

Proof. We precomputed transfer patterns for each optimal cost of the query (Lemma 4.23) and the paths connecting these transfer stations are in our query graph (Lemma 4.24). From the correctly answered direct-connection queries (Lemma 4.21), the time-dependent Dijkstra algorithm on the query graph computes all optimal costs including matching paths. □

4.2.4 Hub Stations

The preprocessing described in Section 4.2.3 uses quadratic time and produces a result of quadratic size. To reduce this, we do these expensive *global* searches only from a suitably preselected set of *hubs* and compute transfer patterns from hubs to all other stations. Note that computing transfer patterns only to other hubs is not faster.

For all non-hub stations, we do *local* searches computing only those transfer patterns without hubs or their parts up to the first hub. More precisely, let $AC_1 \dots C_k B$ be a transfer pattern from a non-hub A that we would have stored in Section 4.2.3. If any of the C_i is a hub, we do not store this pattern any more. The hub C_i with minimal i is called an *access station* of A. We still store transfer patterns $A \dots C_i$ and $C_i \dots B$ into and out of the access station. This shrinks transfer patterns enough to allow query processing entirely from main memory on a single machine, even for large networks. Note that the number of global searches could be reduced further by introducing several levels of hubs, but in our implementation the total cost for the global searches is below the total cost for the local searches already with one level of hubs; see Section 4.2.10.

Selecting the hubs. We create a time-independent graph by overlaying the nodes and edges of each line (as computed in Section 4.2.3), using the minimum of edge costs. Then, we perform cost-limited Dijkstra searches from a random sample of source stations. The stations being on the largest number of shortest paths are chosen as hubs. Note that we experimented with a variety of hub selection strategies, but they showed only little difference with respect to preprocessing time and query graph sizes, and so we stuck with the simplest strategy.

Transfer patterns computation. The *global* search remains as described in Section 4.2.3. The *local* search additionally marks labels stemming from labels at transfer nodes of hubs as *inactive*, and stops as soon as all unsettled labels are inactive [130]. Inactive labels are ignored when the transfer patterns are read off. Before that, inactive labels are needed to dominate non-optimal paths around hubs.

Query graph construction. Processing a query $A@\tau \to B$ looks up the set $\mathscr{X}$ of access stations of A and constructs the query graph from the transfer patterns between the station pairs $\{(A,B)\} \cup (\{A\} \times \mathscr{X}) \cup (\mathscr{X} \times \{B\})$. We construct the query graph as described in Section 4.2.3, but we extract the transfer patterns from a source station to multiple target stations at the same time, as they are stored in the same DAG. This usually speeds up the construction, as some of the transfer patterns share common prefixes. The evaluation of the query graph remains unchanged.

Lemma 4.26 *If c is an optimal cost for the station-to-station query $A@\tau_0 \to B$, then the query graph from A to B contains the transfer pattern of a feasible connection for the query that realizes cost c.*

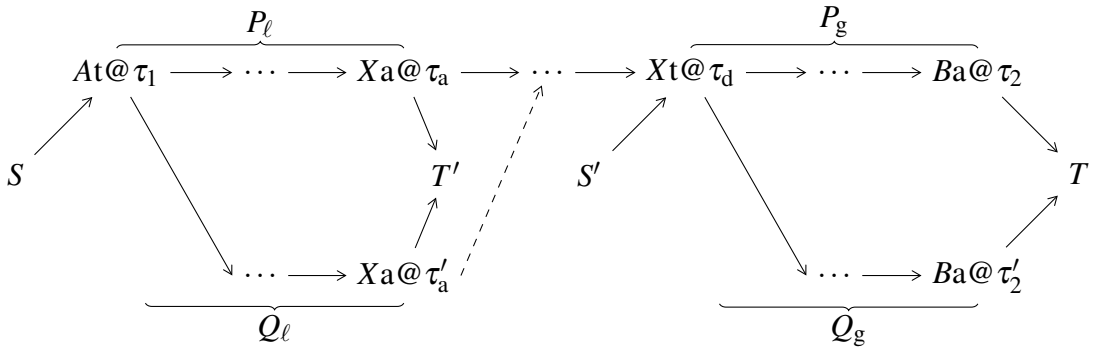


Figure 4.20: From the proof of Lemma 4.26: the optimal S-T-path (top row) that transfers at X, and the S-T'- and S'-T-paths (bottom row) that can be joined (dashed arrow) to an S-T-path of the same optimal cost in the time-expanded graph.

Proof. For a hub A, the global search from A computes the transfer pattern of a connection to B with the optimal cost c (Lemma 4.23) and this transfer pattern is contained in the query graph (Lemma 4.24).

For a non-hub A, we have to show: If the local search from A does not compute the transfer pattern of any connection to B with the optimal cost c, then there is a hub X for which the local search from A computes a transfer pattern $A \dots X$ and the global search from X computes a transfer pattern $X \dots B$ such that there is a connection of cost c with the concatenated transfer pattern $A \dots X \dots B$. The claim then follows from twofold application of Lemma 4.24.

If the local search from A does not compute the transfer pattern of any connection to B of optimal cost c, it instead computes an inactive label for (a prefix of) such a connection. Hence there exist connections to B of cost c that transfer at a hub. Among these, choose one whose first transfer at a hub occurs with the earliest arrival time, and consider the path $S \to P \to T$ representing it in the time-expanded graph from Corollary 4.19 for the query $A@\tau_0 \to B$. Recall that S is the source node at station A and T

is the target node at station B. As depicted in Figure 4.20, we can decompose P into a prefix $P_\ell = \langle A\mathrm{t}@\tau_1, \ldots, X\mathrm{a}@\tau_\mathrm{a}\rangle$ up to the arrival at the first hub X at time τ_a, a suffix $P_\mathrm{g} = \langle X\mathrm{t}@\tau_\mathrm{d}, X\mathrm{d}@\tau_\mathrm{d}, \ldots, B\mathrm{a}@\tau_2\rangle$ from the departure at X at time τ_d onwards, and the transfer piece $P_\mathrm{t} = \langle X\mathrm{a}@\tau_\mathrm{a}, \ldots, X\mathrm{t}@\tau_\mathrm{d}\rangle$ between them.

The searches from A and X do not, in general, find the paths P_ℓ and P_g or their transfer patterns, but others that can be combined to yield the same optimal cost. Therefore, we need a somewhat technical argument to derive them.

To derive what the global search from X computes, we consider the query $X@\tau_\mathrm{d} \to B$ that asks to depart from X no earlier than P. Extending the time-expanded graph for this query per Corollary 4.19 yields a source node S' at X and the same target node T as before. By Lemma 4.23, the global search from X computes the transfer pattern $X \ldots B$ of a path $S' \to Q_\mathrm{g} \to T$ whose cost is better than or equal to the cost of $S' \to P_\mathrm{g} \to T$; in particular, its arrival time τ_2' is no later than the arrival time τ_2 of P_g, and its penalty score is no worse.

Let us now turn to the local search from A, considering the query $A@\tau_0 \to X$ and, per Corollary 4.19, the source node S at A (as before) and a target node T' at X in the time-expanded graph. As P has its transfer at a hub occurring with the earliest arrival time, no connection with cost better than or equal to that of $S \to P_\ell \to T'$ transfers at a hub before reaching $X\mathrm{a}@\tau_\mathrm{a}$. So no inactive label could dominate a label representing P_ℓ at $X\mathrm{a}@\tau_\mathrm{a}$. Therefore, and by reasoning analogous to Lemma 4.23, the local search from A computes the transfer pattern $A \ldots X$ of a path $S \to Q_\ell \to T'$ with cost better than or equal to that of $S \to P_\ell \to T'$. In particular, Q_ℓ arrives at X no later than P_ℓ. Hence there is a path Q_t through the waiting chain of X from the last node $X\mathrm{a}@\tau_\mathrm{a}'$ of Q_ℓ to the first node $X\mathrm{t}@\tau_\mathrm{d}$ of Q_g (the dashed arrow in Figure 4.20).

Let $Q = Q_\ell \to Q_\mathrm{t} \to Q_\mathrm{g}$. It remains to show that the cost of $S \to Q \to T$ is no worse than the cost of $S \to P \to T$ (and then necessarily equal, by optimality of the latter). Duration is no worse because Q_g arrives no later than P_g. Penalty is no worse because P_t and Q_t carry the same transfer penalty of X, and the penalties of Q_ℓ and Q_g, respectively, are no worse than those of P_ℓ and P_g. That proves our claim. □

4.2.5 Walking between Stations

Walking to a nearby station is very important in metropolitan networks. Formally, for each station S we are given a set $\mathcal{N}_S$ of nearby stations where we can walk to. For each $T \in \mathcal{N}_S$, we denote the walking cost by $\mathrm{walk}(S,T)$. The duration includes the minimum transfer duration, as it would be otherwise unclear whether to use the one at S or T. So S is in $\mathcal{N}_S$ with duration $d(\mathrm{walk}(S,S)) = \mathrm{transfer}(S)$ (except if there is never a departure at S). We do not allow walking from S via a station in $\mathcal{N}_S$ to another station not in $\mathcal{N}_S$ (*via walking*). A reason is that via walking often results in detours, as it may not be necessary to walk exactly via this station. Another reason is that we can control the maximum walking distance to increase the comfort of traveling with luggage.

The definition of a *consistent connection* in Section 2.3 is extended to regard $\mathcal{N}_S$. Let $P = (c_1, \ldots, c_k)$ be a sequence of elementary connections. Such a sequence P is called a *consistent connection* from station $S_\mathrm{d}(P)$ to $S_\mathrm{a}(P)$ w. r. t. walking if it fulfills the following two consistency conditions:

1. The departure station $S_\mathrm{d}(c_{i+1})$ of elementary connection c_{i+1} is in the set of nearby stations $\mathcal{N}_{S_\mathrm{a}(c_i)}$ of the arrival station $S_\mathrm{a}(c_i)$ of elementary connection c_i.

2. The walking costs (including the minimum transfer durations) are respected; either $S_\mathrm{d}(c_{i+1}) = S_\mathrm{a}(c_i)$ and $Z_\mathrm{d}(c_{i+1}) = Z_\mathrm{a}(c_i)$, or $\mathrm{dep}_{i+1}(P) - \mathrm{arr}_i(P) \geq d\big(\mathrm{walk}\big(S_\mathrm{a}(c_i), S_\mathrm{d}(c_{i+1})\big)\big)$.

With this definition of a consistent connection, Definition 4.18 describes a station-to-station query that allows to walk on a transfer. Here and in the following, if walking is considered, a consistent connection and a station-to-station query always allow to walk on a transfer.

The construction of the time-expanded graph can be easily adapted to consider walking. Instead of adding transfer edges only within a station S, we add transfer edges respecting $\mathcal{N}_S$: For each arrival node $S\mathrm{a}@\tau$ and each $T \in \mathcal{N}_S$, we put an edge to the first transfer node $T\mathrm{t}@\tau'$ with $\tau' \geq \tau + d(\mathrm{walk}(S,T))$ in the waiting chain of T. We add these edges between arrival nodes and transfer nodes to prohibit via walking. This results in about twice more arcs in the graph and duplicates certain entities in the algorithm. Corollary 4.19 is still valid to answer a station-to-station query in this time-expanded graph.

Here, we will describe how to adapt the basic algorithms of Section 4.2.3. Using walking edges together with hubs will be described separately in Section 4.2.7.

Transfer patterns precomputation. As now two stations can be involved in a single transfer, we need to store both in the transfer pattern. If a transfer happens at a station without walking, this station appears twice in the pattern. A consecutive pair of stations in a transfer pattern now either represents riding a train between these two stations, or walking. As we do not allow via walking, transfer patterns alternate between riding a train and walking. They always begin and end with riding a train, so the number of

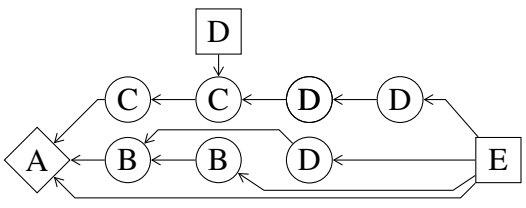


Figure 4.21: DAG for the transfer patterns ‘AE’, ‘ABBE’, ‘ABDE’, ‘ACCD‘ and ‘ACCDDE’. The root node is the diamond, prefix nodes are circles and target nodes are rectangles.

stations in a pattern is always even.[8] Therefore, we can still store them as a simple sequence of stations, without marking walking explicitly. That allows to store them as a DAG (Section 4.2.3), as before. Figure 4.21 shows an example. We can use a slightly modified version of the algorithm *TransferPatterns*(A) of Section 4.2.3 to compute all transfer patterns from A. The only modification is that we now need to store two stations per transfer. Therefore, Lemma 4.27 is a direct consequence of Lemma 4.23.

Lemma 4.27 *If c is an optimal cost for the station-to-station query $A@\tau_0 \to B$, our algorithm computes the transfer pattern of a feasible connection for the query that realizes cost c.*

Query graph construction and evaluation. As we prohibit via walking, not only the cost of reaching a station is important, but also whether we walked to the station, or not (and can thus walk to nearby stations). To distinguish these labels, our query graph can contain now up to two nodes per station S, one *departure node* S^{d} and one *arrival node* S^{a}.

An edge from a departure node to an arrival node represents riding a train, whereas an edge from an arrival node to a departure node represents a transfer (potentially between different stations). There are no edges between two arrival nodes or two departure nodes. We build the query graph from a DAG as in Section 4.2.3, but we now alternate between arrival node and departure node. As we build it backwards, we start with an arrival node. Figure 4.22 shows the query graph from A to E built from the DAG in Figure 4.21.

Lemma 4.28 *For each transfer pattern $AC_1C_2\dots C_{k-1}C_kB$ in the DAG there exists the path $\langle A^{\mathrm{d}}, C_1^{\mathrm{a}}, C_2^{\mathrm{d}} \dots, C_{k-1}^{\mathrm{a}}, C_k^{\mathrm{d}}, B^{\mathrm{a}}\rangle$ in the constructed query graph.*

Proof. Let $AC_1\dots C_kB$ be a transfer pattern in the DAG. We know that k is even. Induction over i shows that for even i there is a path $\langle C_i^{\mathrm{d}}, \dots, C_k^{\mathrm{d}}, B^{\mathrm{a}}\rangle$ in the query graph, and the node C_i^{d} in the query graph gets processed together with the prefix node '$AC_1\dots C_{i-1}$' in the DAG. As we alternate between arrival and departure nodes, we add edge $(C_{i-1}^{\mathrm{a}}, C_i^{\mathrm{d}})$. For odd i there is the path $\langle C_i^{\mathrm{a}}, \dots, C_k^{\mathrm{d}}, B^{\mathrm{a}}\rangle$ in the query graph. Finally, node C_1^{a} is processed with the root node 'A' in the graph, so that also the edge (A, C_1^{a}) is added and the path $\langle A^{\mathrm{d}}, C_1^{\mathrm{a}}, \dots, C_k^{\mathrm{d}}, B^{\mathrm{a}}\rangle$ exists in the query graph. □

The evaluation of the query graph is done similar to Section 4.2.3. We use a time-dependent multi-criteria Dijkstra search on the query graph. An edge from a departure node to an arrival node is evaluated as before using a direction-connection query. An edge from an arrival node to a departure node is evaluated by a simple look-up into a table that stores the (time-independent) transfer and walking costs between the stations.

Theorem 4.29 *For a given query $A@\tau \to B$, the described search on the query graph from A^{d} to B^{a} returns the set of optimal costs and for each such cost a corresponding path.*

[8] We will see in Section 4.2.7 that the decomposition of transfer patterns using hubs changes this.

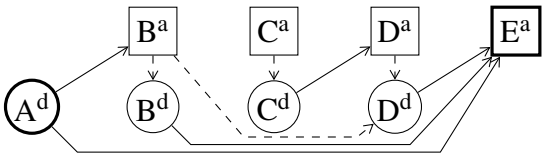


Figure 4.22: Query graph A→ E from transfer patterns 'AE', 'ABBE', 'ABDE' and 'ACCDDE'. Departure nodes are circles and arrival nodes are rectangles. Walking edges are dashed.

Proof. We precomputed transfer patterns for each optimal cost of the query (Lemma 4.27) and the paths connecting these transfer stations are in our query graph between A^{d} to B^{a} (Lemma 4.28). From the correctly answered direct-connection queries (Lemma 4.21) and walking queries, the time-dependent Dijkstra algorithm on the query graph computes all optimal costs including matching paths. □

4.2.6 Location-to-Location Query

Our implementation is able to answer location-to-location queries (Definition 4.30). This feature is of high practical value for dense metropolitan networks, but, to the best of our knowledge, disregarded in all earlier work.

Definition 4.30 *A* location-to-location query *is given as $Z@\tau \to Z'$, where Z is the source location, τ is the earliest departure time, and Z' is the target location. Each station also has a location assigned. Locations are given in some geographic coordinate system. The walking cost (duration and penalty) between two locations Z_1 and Z_2 is given by an oracle*[9] $\mathrm{walk_o}(Z_1, Z_2)$. *Let $\mathscr{R}_Z$ be the set of stations within a reasonable walking distance of location Z. All consistent connections from a station $A \in \mathscr{R}_Z$ to a station $B \in \mathscr{R}_{Z'}$ that do not depart earlier than $\tau + d(\mathrm{walk_o}(Z,A))$ are* feasible *for this query, but their cost is increased by the walking costs* $\mathrm{walk_o}(Z,A)$, $\mathrm{walk_o}(B,Z')$, *and the waiting time until the connection departs at A.*

We define optimal connections *and* optimal costs *just as in Definition 4.18 of station-to-station queries.*

We can compute the result of a location-to-location query in the time-expanded graph using Corollary 4.31.

Corollary 4.31 *Consider a location-to-location query $Z@\tau \to Z'$. The extended time-expanded graph for this query has a source node Z with outgoing edges to all stations $A \in \mathscr{R}_Z$. More precisely, the edge to A points to the first transfer node $A\mathrm{t}@\tau'$ such that τ' is no less than τ plus the walking duration to A; its cost has duration $\tau' - \tau$ and*

[9] In reality, we may approximate these costs via the straight-line distance between the respective locations, or we have a separate network of footpaths on which we compute shortest paths.

the walking penalty. Likewise, the arrival nodes of all stations $B \in \mathscr{R}_{Z'}$ are connected to the target node Z' with edges that carry the walking cost from B.

Exactly the paths from Z to Z' are the feasible connections for this query. Each of these paths has the cost of the feasible connection it represents, including the walking costs from Z and to Z'.

Note that we explicitly distinguish between walking from the source location and to the target location of a location-to-location query, and walking involved in a transfer (Section 4.2.5). In particular, the set of locations $\mathscr{R}_S$ within reasonable walking distance of the location of station S is usually larger than $\mathscr{N}_S$. There are several reasons for that: Quality of the query result is a concern, as sometimes only a reasonable connection starting quite far away from the current location is available. Also, people tend to accept less walking when they are on a journey than in the beginning or the end, as they potentially have a car available there. The third reason is a practical one, a larger $\mathscr{N}_S$ would make the time-expanded graph larger and thus the precomputation more expensive.

Query graph construction and evaluation. For a location-to-location query $Z@\tau \rightarrow Z'$, the query graph is built from transfer patterns for all pairs of source and target stations in $\mathscr{R}_Z \times \mathscr{R}_{Z'}$. As source and target are no stations, we need to add separate nodes for Z and Z' to the query graph. For each station $A \in \mathscr{R}_Z$, we add an edge from Z to A^{d} of cost $\mathrm{walk}_{\mathrm{o}}(Z,A)$. And for each station $B \in \mathscr{R}_{Z'}$, we add an edge from B^{a} to Z' of cost $\mathrm{walk}_{\mathrm{o}}(B,Z')$. Figure 4.23 shows an example.

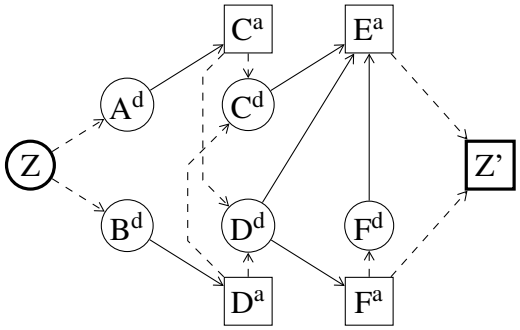


Figure 4.23: Query graph Z→ Z' from $\mathscr{R}_{\mathrm{Z}} = \{\mathrm{A},\mathrm{B}\}$, $\mathscr{R}_{\mathrm{Z}'} = \{\mathrm{E},\mathrm{F}\}$, and transfer patterns 'ACCE', 'ACDE', 'ACDF', 'BDDE', 'BDCE', 'BDDFFE' and 'BDDF'. Departure nodes are circles and arrival nodes are rectangles. Walking edges are dashed.

We construct the query graph as described in Section 4.2.5, but combine it with the idea of Section 4.2.4 to extract the transfer patterns from a source station to multiple target stations at the same time, as they are stored in the same DAG. The complete construction works as follows:

1. For each $A \in \mathscr{R}_Z$, add an edge (Z,A^{d}).

2. For each $B \in \mathscr{R}_{Z'}$, add an edge (B^{a}, Z').
3. Fetch the precomputed transfer patterns DAG for a station $A \in \mathscr{R}_Z$.
4. For all $B \in \mathscr{R}_{Z'}$, search the corresponding target node x in the DAG and put (x, B^{a}) in a set $\mathscr{S}$. Keep a mark for each prefix node of the DAG, initially all nodes are unmarked.
5. Remove a (x, B^y) from the set $\mathscr{S}$. Let $y' := \mathrm{a}$ if $y = \mathrm{d}$, otherwise $y' := \mathrm{d}$. Assume node x has ℓ unmarked successor prefix nodes $x_1, \ldots, x_\ell$ with labels $C_1, \ldots, C_\ell$. Add the edges $(C_1^{y'}, B^y), \ldots, (C_\ell^{y'}, B^y)$ to the query graph. Set the mark for $x_1, \ldots, x_\ell$, and add $(x_1, C_1^{y'}), \ldots, (x_\ell, C_\ell^{y'})$ to $\mathscr{S}$. If x has A as successor node, add the edge $(A^{\mathrm{d}}, C^{\mathrm{a}})$ to the query graph.
6. Recursively perform Step 5 until the set $\mathscr{S}$ is empty.
7. Repeat from Step 3 until all stations $A \in \mathscr{R}_Z$ are processed.

The evaluation of the query graph is done as described in Section 4.2.5, with the edges incident to Z and Z' being evaluated using a look-up into a table that stores the (time-independent) transfer and walking costs given by the oracle $\mathrm{walk}_{\mathrm{o}}$.

Theorem 4.32 *For a given location-to-location query $Z@\tau \to Z'$, the described search on the query graph from Z to Z' returns the set of optimal costs and for each such cost a corresponding path.*

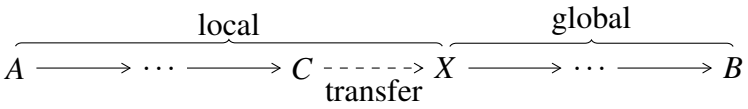
Proof. Every optimal connection of the location-to-location query $Z@\tau \to Z'$ also describes an optimal connection for a station-to-station query $A@\tau' \to B$ with $A \in \mathscr{R}_Z$, $B \in \mathscr{R}_{Z'}$ and $\tau' = \tau + \mathrm{walk}_{\mathrm{o}}(Z, A)$. Therefore, a transfer pattern for the cost of the optimal connection was precomputed (Lemma 4.27) and added to our query graph. Following the proof of Theorem 4.29, and the correctly assigned costs to the edges incident to Z and Z', the time-dependent Dijkstra algorithm on the query graph computes all optimal costs including matching paths. □

4.2.7 Walking and Hubs

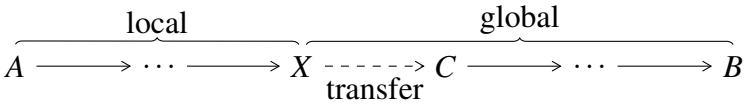
Extending transfer patterns, the query graph construction and evaluation to support walking was more or less straightforward. Improving the preprocessing by using hubs is more difficult. A transfer can now involve two stations and we need to distinguish the case where the arrival station is a hub or the departure station is a hub. In Section 4.2.4 we assumed that both stations are the same and exploited this to get a clean decomposition of a transfer pattern into a local and a global part as depicted in Figure 4.24(a). It is clean in the sense that both parts are a transfer pattern that begins and ends with riding a train. The local part ended with the arrival at the first hub X where a transfer occurred,

and the global part began with the departure at X. As arrival and departure happened at the same station, the transfer at station X was implicitly given. But with walking edges, there is no clean decomposition anymore as we now need to store the transfer explicitly. In case that the departure station is a hub (Figure 4.24(b)), the local search needs to compute a pattern ending with the transfer to X. And in case that the arrival station is a hub (Figure 4.24(c)), the global part needs to start with a transfer from X.

(a) Without walking



(b) With walking, case 1: departure at hub X



(c) With walking, case 2: arrival at hub X

Figure 4.24: Decomposition of a transfer pattern at the first hub X into the part computed by the local search and the part computed by the global search.

The local search can be extended efficiently to compute the transfer patterns ending with an arrival at the first hub (*arrival access station*), and the ones ending with a transfer at the first hub (*departure access station*). But there is a separate global search necessary, one for each case, making the precomputation significantly more expensive.

We could ignore the second case by requiring that a hub must always be a departure station, as depicted in Figure 4.24(b). But this limits the effectiveness of the local search, as we need to compute and store more transfer patterns, or need to increase the number of hubs. We will illustrate this with Example 4.33.

Example 4.33 *Consider the case that we only decompose a transfer pattern at a departure at a hub. Assume that a smaller city has a main station where all the long distance trains arrive, and some bus stops in front. Each connection into the city arrives at the main station and transfers to one of the bus stops. We could therefore not decompose the transfer pattern of such a connection, and a local search from outside the city could therefore not compute transfer patterns just to the main station, but into the whole city. We could fix this by making all bus stops to hubs, but this significantly increases the number of global searches.*

Transfer patterns precomputation. As just mentioned, we need two types of global searches, a *global departure search* and a *global arrival search*. The global departure search from a hub X remains as the normal transfer patterns computation algorithm described in Section 4.2.5. It computes transfer patterns starting with a departure at hub X. The global arrival search computes transfer patterns starting with a transfer from X. Definition 4.34 defines the query for a connection starting with a transfer. It can be answered in the time-expanded graph using Corollary 4.35.

Definition 4.34 *A* connecting arrival query $X@\tau \to B$ *has a source hub X, an arrival time τ at X, and a target station B.*

All consistent connections from a station $C \in \mathcal{N}_X$ to station B that do not depart earlier than $\tau + d(\text{walk}(X,C))$ are feasible *for this query, but their cost is increased by the walking cost* $\text{walk}(X,C)$ *and the waiting time until the connection departs at C. We define* optimal connections *and* optimal costs *just as in Definition 4.18 of station-to-station queries.*

Corollary 4.35 *Consider a* connecting arrival query $X@\tau \to B$*. For this query, we extend the time-expanded graph by a source node S and a target node T. For all $C \in \mathcal{N}_X$, take the first transfer node $C\text{t}@\tau'$ with $\tau' \geq \tau + d(\text{walk}(X,C))$. We add an edge of duration $\tau' - \tau$ and penalty $p(\text{walk}(X,C))$ that leads to $C\text{t}@\tau'$. Also add edges of cost zero from all arrival nodes of B to target node T.*

Exactly the paths from S to T are the feasible connections *for the query. Each of these paths has the cost of the feasible connection it represents, including the walking cost from X.*

To compute transfer patterns starting with a transfer from X, we use algorithm *TransferPatternsArrival(X)*:

1. Run a multi-criteria variant of Dijkstra's algorithm [105, 134, 100] starting at all *transfer* nodes of $C \in \mathcal{N}_X$ from labels of the walking cost $\text{walk}(X,C)$.

2. For every station B, choose optimal connections with the *arrival chain algorithm*: For all distinct arrival times $\tau_1 < \tau_2 < \ldots$ at B, select a dominant subset in the set of labels consisting of (i) those settled at the arrival node(s) at time τ_i and (ii) those selected for time τ_{i-1}, with duration increased by $\tau_i - \tau_{i-1}$; ties to be broken in preference of (ii). Note that this step is identical to Step 2 of algorithm TransferPatterns(X) in Section 4.2.3.

3. Trace back the paths of all labels selected in Step 2. Create the DAG of transfer patterns of these paths under a common root node X by prepending station X to each transfer pattern. Do this by traversing the labels in the order in which they have been settled.

To distinguish between the DAG computed by the global departure search and the global arrival search from a hub X, the first one has a root node X^{d} and the second one a root node X^{a}.

Lemma 4.36 proves that our global arrival search computes the corresponding optimal transfer patterns.

Lemma 4.36 *If c is an optimal cost for the connecting arrival query $X@\tau_0 \to B$, TransferPatternsArrival(X) computes the transfer pattern of a feasible connection for the query that realizes cost c, prepended with station X.*

Proof. By definition, the optimal cost c is the cost of an optimal path $S \to P_1 \to T$ in the extended time-expanded graph from Corollary 4.35, where P_1 starts at some node $C\mathrm{t}@\tau_{\mathrm{d}}$ with $C \in \mathcal{N}_X$ (the successor of S) and ends at some node $B\mathrm{a}@\tau_{\mathrm{a}}$.

We will now attach alternative source and target nodes S' and T' to the graph that reflect the transfer patterns computation.

For all $C' \in \mathcal{N}_X$, S' has an edge of cost $\mathrm{walk}(X,C')$ to all transfer nodes of C'. This reflects the initial labels. All arrival nodes $B\mathrm{a}@\tau$ with $\tau \le \tau_{\mathrm{a}}$ have an edge of duration $\tau_{\mathrm{a}} - \tau$ and penalty 0 to T'. Hence T' corresponds to the label set for arrival time τ_{a}. A transfer pattern is computed for a path P_2' such that $S' \to P_2' \to T'$ has better or equal cost than $S' \to P_1 \to T'$. P_2' starts at a node $C'\mathrm{t}@\tau_{\mathrm{d}}'$ with $C' \in \mathcal{N}_X$. In particular, we need to start walking from X to reach P_2' no earlier than to reach P_1. So we can prepend to P_2' the part of the waiting chain of C' between the first node $C'\mathrm{t}@\tau'$ with $\tau' \ge \tau_0 + d(\mathrm{walk}(X,C'))$ and $C'\mathrm{t}@\tau_{\mathrm{d}}'$; let P_2 denote this extension of P_2'.

To prove that P_2 is the claimed path, it remains to show that $S \to P_1 \to T$ and $S \to P_2 \to T$ have the same cost. By construction of S', T' and by choice of P_2', the following inequalities hold for duration and penalty of the paths:

$$d(S \to P_1 \to T) = \tau_{\mathrm{a}} - \tau_0 = d(S \to P_2 \to T') \ge d(S \to P_2 \to T),$$
$$p(S \to P_1 \to T) = p(S' \to P_1 \to T') \ge p(S' \to P_2' \to T') = p(S \to P_2 \to T).$$

As $S \to P_1 \to T$ is optimal, equality holds throughout. □

We also need to extend the *local search* to compute transfer patterns ending with a transfer to a hub. Currently, the arrival chain algorithm (Section 4.2.3) selects only labels that arrive at a station, and therefore the transfer pattern ends with riding a train. Definition 4.37 defines the query for a connection ending with a transfer. To answer such a query in the time-expanded graph, Corollary 4.38 is used.

Definition 4.37 *A* local search walking query *$A@\tau \to X$ has a source station A, an earliest departure time τ and a target hub X.*

All consistent connections from station A to a station C with $X \in \mathcal{N}_C$ that do not depart earlier than τ are feasible *for this query, but their cost is increased by the waiting time until the connection departs at A, the walking cost* $\mathrm{walk}(C,X)$ *plus the waiting time to the earliest departing connection at X. If there is no such departing connection at X,*

then the connection is not feasible. We define optimal connections *and* optimal costs *just as in Definition 4.18 of station-to-station queries.*

Corollary 4.38 *Consider a* local search walking query $A@\tau \to X$. *Take the first transfer node* $A\text{t}@\tau'$ *with* $\tau' \geq \tau$. *For this query, we extend the time-expanded graph by a source node S with an edge of duration* $\tau' - \tau$ *and penalty* 0 *that leads to* $A\text{t}@\tau'$ *and by a target node T with incoming edges of zero cost from all transfer nodes of X.*

Exactly the paths from S to T are the feasible connections for the query. Each of these paths has the cost of the feasible connection it represents, including the walking cost to X and the waiting time to the earliest departing connection at X.

To get transfer patterns ending with riding a train *and* ending with a transfer to a hub, we use algorithm *LocalSearchWalk(A)*:

1. Run a multi-criteria variant of Dijkstra's algorithm [105, 134, 100] starting from labels of cost zero at all *transfer* nodes of station A. Mark labels stemming from labels at transfer nodes of hubs as inactive (as in Section 4.2.4). Further mark labels at transfer nodes with their direct parent being at an arrival node of a hub as inactive. A label is always inactive if its parent is inactive. Stop as soon as all unsettled labels are inactive.

2a. For every station B with active labels, choose optimal connections with the *arrival chain algorithm*: For all distinct arrival times $\tau_1 < \tau_2 < \ldots$ at B, select a dominant subset in the set of labels consisting of (i) those settled at the arrival node(s) at time τ_i and (ii) those selected for time τ_{i-1}, with duration increased by $\tau_i - \tau_{i-1}$; ties to be broken in preference of (ii).

2b. For every hub X with active labels, choose optimal departing connections with the *departure chain algorithm*: For all distinct departure times $\tau_1 < \tau_2 < \ldots$ at X, select a dominant subset in the set of labels consisting of (i) those settled at transfer node(s) at time τ_i, (ii) those settled at an arrival node of X having an edge to a transfer node $X\text{t}@\tau_i$, and (iii) those selected for time τ_{i-1} with duration increased by $\tau_i - \tau_{i-1}$; ties to be broken in preference of (ii).

3. Trace back the paths of all labels selected in Steps 2a and 2b. Create the DAG of transfer patterns of these paths by traversing them from the source A.

Note that contrary to the arrival chain, the departure chain already exists in the graph as the waiting chain (with potentially multiple transfer nodes per departure time), so that we could use the multi-criteria variant of Dijkstra's algorithm to compute the dominant connections. However, the propagation through the whole chain can be incomplete, as we stop the search once all unsettled labels are inactive.

To store all transfer patterns of a local search from a station A in a single DAG, we need to distinguish between transfer patterns that end with riding a train and that end

with a transfer. Transfer patterns that end with riding a train at a station S have a target node S^{a}, the others ending with a transfer have a target node S^{d}, see Figure 4.25. The root node is always A^{d}, as all transfer patterns begin with riding a train.

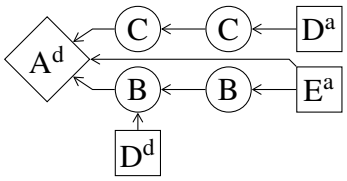


Figure 4.25: DAG for the transfer patterns 'AE', 'ABBE', 'ABD', and 'ACCD'. The root node is the diamond, prefix nodes are circles and target nodes are rectangles. A target node S^{a} denotes an arrival as the transfer pattern ends with riding a train, and a target node S^{d} denotes a departure as the transfer pattern ends with a transfer.

Lemma 4.39 proves that our extended local search computes the corresponding optimal transfer patterns.

Lemma 4.39 *If c is an optimal cost for the local search walking query $A@\tau_0 \to X$, and there is no feasible connection with cost c that has a transfer at a hub before reaching a transfer node at X (in particular, does not arrive at X), then LocalSearchWalk(A) computes the transfer pattern of a feasible connection for the query that realizes cost c.*

Proof. By definition, the optimal cost c is the cost of an optimal path $S \to P_1 \to T$ in the extended time-expanded graph from Corollary 4.38, where P_1 starts at some node $A\mathrm{t}@\tau_1$ (the successor of S) and ends at some node $X\mathrm{t}@\tau_2$.

We will now attach alternative source and target nodes S' and T' to the graph that reflect the transfer patterns computation with LocalSearchWalk(A). S' has an edge of duration 0 and penalty 0 to all transfer nodes of A. This reflects the initial labels. All transfer nodes $X\mathrm{t}@\tau$ with $\tau \leq \tau_2$ have an edge of duration $\tau_2 - \tau$ and penalty 0 to T'. Hence T' corresponds to the label set for departure time τ_2 in the departure waiting chain. By definition, there is no feasible connection with cost c that has a transfer at a hub before reaching a transfer node at X. Therefore, a label at T' that would represent path P_1 could not be dominated by an inactive label. But as there is potentially an active label dominating such a label (or a predecessor), we only know that there is a dominant label for a path P_2' such that $S' \to P_2' \to T'$ has better or equal cost than $S' \to P_1 \to T'$.

In particular, P_2' departs from A no earlier than P_1 does. That means, we can prepend to P_2' the part of the waiting chain of A between the first node of P_1 and the first node of P_2'; let P_2 denote this extension of P_2'.

To prove that P_2 is the claimed path, it remains to show that $S \to P_1 \to T$ and $S \to P_2 \to T$ have the same cost. By construction of S', T' and by choice of P_2', the following inequalities hold for duration and penalty of the paths:

$$d(S \to P_1 \to T) = \tau_{\mathrm{a}} - \tau_0 = d(S \to P_2 \to T') \geq d(S \to P_2 \to T),$$
$$p(S \to P_1 \to T) = p(S' \to P_1 \to T') \geq p(S' \to P_2' \to T') = p(S \to P_2 \to T).$$

By optimality of $S \to P_1 \to T$, equality holds throughout. □

Query graph construction. We combine the query graph construction algorithms of Section 4.2.4 and Section 4.2.5. For a query $A@\tau \to B$, we look-up the set $\mathscr{X}$ of access stations of A. Note that a hub X may occur twice in $\mathscr{X}$, once as arrival access station X^{a} and once as departure access station X^{d}. We construct the query graph from the transfer patterns stored in the precomputed DAGs. The pairs of (root node, target node) where the transfer patterns are stored are $\{(A^{\mathrm{d}}, B^{\mathrm{a}})\} \cup (\{A^{\mathrm{d}}\} \times \mathscr{X}) \cup (\mathscr{X} \times \{B^{\mathrm{a}}\})$. The evaluation of the query graph remains as described in Section 4.2.5. Therefore, Lemma 4.40 is sufficient to prove the correctness of our query.

Lemma 4.40 *If c is an optimal cost for the station-to-station query $A@\tau_0 \to B$, then the query graph from A to B contains the transfer pattern of a feasible connection for the query that realizes cost c.*

Proof. For a hub A, the global search from A computes the transfer pattern of a connection to B with the optimal cost c (Lemma 4.27) and this transfer pattern is contained in the query graph (Lemma 4.28).

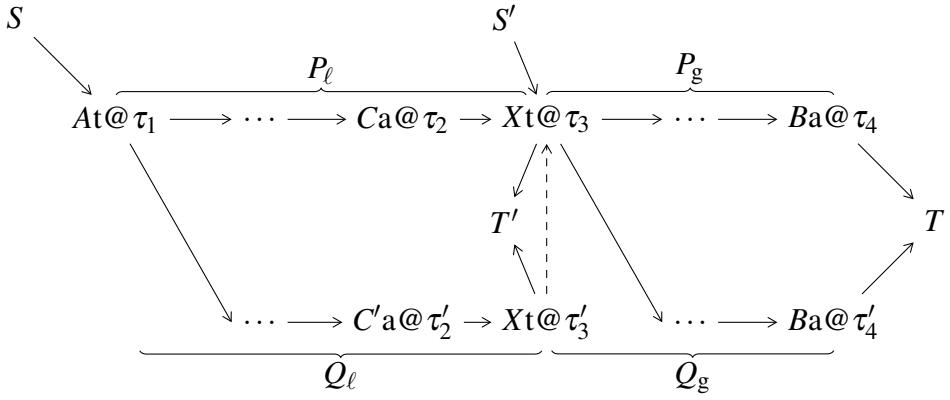
For a non-hub A, we have to show: If the local search from A does not compute the transfer pattern of any connection to B with the optimal cost c, then there is a hub X for which the local search from A computes a transfer pattern $A \ldots X$ and a global search from X computes a transfer pattern $X \ldots B$ such that there is a connection of cost c with the concatenated transfer pattern $A \ldots X \ldots B$.

If the local search from A does not compute the transfer pattern of any connection to B of optimal cost c, it instead computes an inactive label for (a prefix of) such a connection. Hence there exist connections to B of cost c that transfer at a hub. Among these, choose one whose first transfer at a hub occurs with the earliest time at the hub, and consider the path $S \to P \to T$ representing it in the time-expanded graph from Corollary 4.19 for the query $A@\tau_0 \to B$. Recall that S is the source node at station A and T is the target node at station B. We need to distinguish between X as a departure access station and an arrival access station.

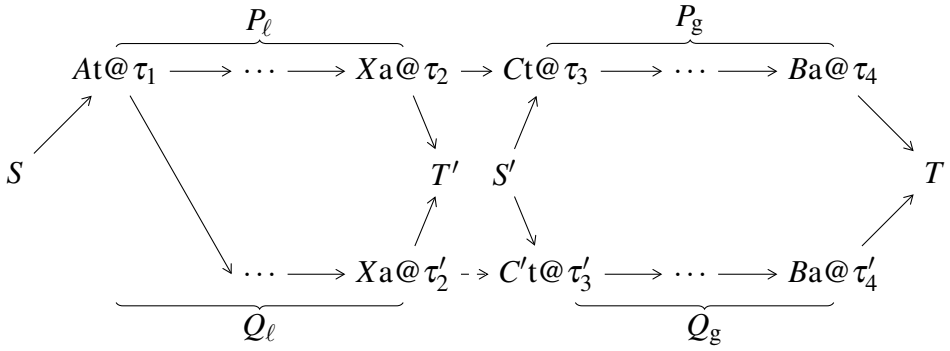
Case 1: X is a *departure* access station. As depicted in Figure 4.26(a), we can decompose P into $P_\ell = \langle A\mathrm{t}@\tau_1, \ldots, C\mathrm{a}@\tau_2, X\mathrm{t}@\tau_3 \rangle$ up to the first transfer node at station X at time τ_3, and $P_{\mathrm{g}} = \langle X\mathrm{t}@\tau_3, \ldots, B\mathrm{a}@\tau_4 \rangle$ till the arrival at station B.

The local search from A and the global departure search from X do not, in general, find the paths P_ℓ and P_{g} or their transfer patterns, but others that can be combined to yield the same optimal cost.

To derive what the global departure search from X computes, we consider the *station-to-station query* $X@\tau_3 \to B$ that asks to depart from X no earlier than P. Extending the time-expanded graph for this query per Corollary 4.19 yields a source node S' at X and the same target node T as before. By Lemma 4.27 the global departure search from X computes the transfer pattern $X \ldots B$ of a path $S' \to Q_{\mathrm{g}} \to T$ whose cost is better than or



(a) Case 1: departure at hub X



(b) Case 2: arrival at hub X

Figure 4.26: From the proof of Lemma 4.40: In each subfigure are the optimal S-T-path (top row), and the S-T'- and S'-T-paths (bottom row) that can be joined (dashed arrow) to an S-T-path of the same optimal cost in the time-expanded graph.

equal to the cost of $S' \to P_g \to T$; in particular, arrival time τ'_4 is no later than the arrival time τ_4 of P_g, and its penalty score is no worse.

Let us now turn to the local search from A, considering the *local search walking query* $A@\tau_0 \to X$ and, per Corollary 4.38, the source node S at A (as before) and a target node T' at X in the extended time-dependent graph. As P has its transfer at a hub occurring with the earliest arrival time, no connection with cost better than or equal to that of $S \to P_\ell \to T'$ transfers at a hub before reaching $X\mathrm{t}@\tau_3$. Therefore, by Lemma 4.39, the local search from A computes the transfer pattern $A \ldots X$ of a path $S \to Q_\ell \to T'$ with cost better than or equal to that of $S \to P_\ell \to T'$. In particular, Q_ℓ arrives at X no later than P_ℓ. Hence there is a path Q_t through the waiting chain of X from the last node $X\mathrm{t}@\tau'_3$ of Q_ℓ to the first node $X\mathrm{t}@\tau_3$ of Q_g (the dashed arrow in Figure 4.26(a)). Note that Q_ℓ can arrive at a station C' that is different from the arrival station C of P_ℓ. But as we only care about a timely transfer to X, this is no problem.

Let $Q = Q_\ell \to Q_\mathrm{t} \to Q_\mathrm{g}$. It remains to show that the cost of $S \to Q \to T$ is no worse than the cost of $S \to P \to T$ (and then necessarily equal, by optimality of the latter). Duration is no worse because Q_g arrives no later than P_g. Penalty is no worse because Q_t carries no penalty as it only represents waiting at X, and the penalties of Q_ℓ and Q_g, respectively, are no worse than those of P_ℓ and P_g.

Case 2: X is an *arrival* access station. As depicted in Figure 4.26(b), we can decompose P into a prefix $P_\ell = \langle A\mathrm{t}@\tau_1, \ldots, X\mathrm{a}@\tau_2 \rangle$ up to the arrival node at station X at time τ_2, a suffix $P_\mathrm{g} = \langle C\mathrm{t}@\tau_3, \ldots, B\mathrm{a}@\tau_4 \rangle$ from the first transfer node at C onwards, and the transfer piece $P_\mathrm{t} = \langle X\mathrm{a}@\tau_2, C\mathrm{t}@\tau_3 \rangle$ between them.

Again, the local search from A and the global arrival search from X do not, in general, find the paths P_ℓ and $P_\mathrm{t} \to P_\mathrm{g}$ or their transfer patterns, but others that can be combined to yield the same optimal cost.

To derive what the global arrival search from X computes, we consider the *connecting arrival query* $X@\tau_2 \to B$ that asks for a connection after arriving at time τ_2 at X. Extending the time-expanded graph for this query per Corollary 4.35 yields a source node S' with edges to all nearby stations $C \in \mathcal{N}_X$ and the same target node T as before. Note that the cost of the path $\langle S', C\mathrm{t}@\tau_3 \rangle$ is the same as the cost of P_t. By Lemma 4.36 the global arrival search from X computes the transfer pattern $C' \ldots B$ of a path $S' \to Q_\mathrm{g} \to T$ prepended by station X whose cost is better than or equal to the cost of $S' \to P_\mathrm{g} \to T$; in particular, the cost includes walking from station X at time τ_2 to station C', the arrival time τ_4' is no later than the arrival time τ_4 of P_g, and its penalty score is no worse.

Let us now turn to the local search from A, considering the *station-to-station query* $A@\tau_0 \to X$ and, per Corollary 4.19, the source node S at A (as before) and a target node T' at X in the time-expanded graph. As P has its transfer at a hub occurring with the earliest arrival time, no connection with cost better than or equal to that of $S \to P_\ell \to T'$ transfers at a hub before reaching $X\mathrm{a}@\tau_2$. So no inactive label could dominate a label representing P_ℓ at $X\mathrm{a}@\tau_2$. Therefore, and by reasoning analogous to Lemmas 4.23 and 4.27 the local search from A computes the transfer pattern $A \ldots X$ of a path $S \to Q_\ell \to T'$ with cost better than or equal to that of $S \to P_\ell \to T'$. In particular, Q_ℓ arrives at X no later than P_ℓ. Hence there is an edge to a transfer node $C'\mathrm{t}@\tau_3''$ with $\tau_3'' \leq \tau_3'$. Therefore, there is a path Q_t from $X\mathrm{a}@\tau_2'$ through the waiting chain of C' to the first node $C'\mathrm{t}@\tau_3'$ of Q_g (the dashed arrow in Figure 4.26(b)).

Let $Q = Q_\ell \to Q_\mathrm{t} \to Q_\mathrm{g}$. It remains to show that the cost of $S \to Q \to T$ is no worse than the cost of $S \to P \to T$ (and then necessarily equal, by optimality of the latter). Duration is no worse because Q_g arrives no later than P_g. Penalty is no worse because the penalty of Q_ℓ is no worse than the one of P_ℓ, and the penalty of $Q_\mathrm{t} \to Q_\mathrm{g}$ is the penalty of $S' \to Q_\mathrm{g} \to T$ and is therefore no worse than the penalty of $P_\mathrm{t} \to P_\mathrm{g}$. That proves our claim. □

The query described here answers a station-to-station query (Definition 4.18). To answer a location-to-location query (Definition 4.30) with hubs, compared to Section 4.2.6, the creation of the query graph has to be adapted to consider hub stations as described in this section.

4.2.8 Further Refinements

In the previous sections, we simplified the presentation of our algorithm. Our actual implementation includes the following refinements.

More compact graph model. In the precomputation, we optimize the representation of the graph from Section 2.3.2 in two ways. Departure nodes are removed and their predecessors (transfer node and maybe arrival node) are linked directly to their successor (the next arrival node), cf. [121, §8.1.2]. To exploit the periodicity of timetables, we *roll up* the graph modulo one day, that is, we label nodes with times modulo 24 hours and use bit masks to indicate each connection's traffic days.

Query graph search. After we have determined the earliest arrival time at the target station, we execute a backward search to find the optimal connection that *departs latest*, see Footnote 6 on Page 82.

4.2.9 Heuristic Optimizations

The system described so far gives exact results, that is, for each query we get an optimal connection for every optimal cost. However, despite the use of hubs (Sections 4.2.4 and 4.2.7), the precomputation is not significantly faster than the quadratic precomputation described in Sections 4.2.3 and 4.2.5. The reason is that, although the results of the local searches (the local transfer patterns) are reasonably small, almost every local search has a local path of very large cost and hence has to visit a large portion of the whole network before it can stop. A typical example is an overnight connection to a nearby village for a departure time right after the last bus for the day to that village has left. Such a connection can easily take, say, 15 hours, and in order to compute it, a large fraction of the whole network has to be searched. This *15 hours to the nearby village problem* is actually at the core of what makes precomputing a public transportation network so hard [9].

The good news is that with our transfer patterns approach we don't have this problem at query time but only in the precomputation. Note here that our approach is unique in that it precomputes information for *all* queries, not just for queries where source and target are sufficiently "far apart". The bad news is that, despite intensive thought, we did not find a solution that is both fast and exact. We eventually resorted to the following simple but approximate solution: limit the local searches to at most two transfers, that is, using at most three trains. This is related to the stall-in-advance technique of highway-node routing [130]. We call this the *3-legs heuristic*, and as we will see in Section 4.2.10, it indeed makes the local searches reasonably fast. Theoretically, we may now miss some optimal transfer patterns, but we found this to play no role in the practical use of our algorithm. For example, on our PT-CH graph (Section 4.2.10), on 10 000 random queries the 3-leg heuristic gave only three non-optimal results, and all three of these

were only a few percent off the optimum. We remark that errors in the input data are a much bigger issue in practice.

Having accepted a small fraction of non-optimal results, we also developed and apply various other heuristics, which may lead to a non-optimal solution at query time, but whose measured effect in practice is again tolerable. Together, they speed up our query times by a factor of 3–5. But they are not essential for the feasibility of our approach.

Tightening the dominance relation. To reduce the number of dominant labels at a node during precomputation, we use the idea of *relaxed* Pareto dominance [108, 109]. This idea was initially developed to compute *more labels*, so that also slightly sub-optimal but still reasonable connections are found, but we use it to compute *less labels*. We change our dominance relation so that cost (d_1, p_1) dominates cost (d_2, p_2) iff $d_1 + k \cdot (p_1 - p_2) < d_2$ for a relaxation factor $k \geq 0$. A label with larger duration is only dominated if its penalty is not sufficiently smaller. In a sense, we only optimize duration, but relax it by penalty. That way, we actually *tighten* the previous dominance relation that regards both duration and penalty in the Pareto sense. We had a choice of relaxing duration by penalty, or penalty by duration. But as duration (contrary to penalty) decides on the consistency of a connection (we may miss a connecting train) we chose the first.

The value of k lets us choose the tightness of our dominance relation. The smaller k, the fewer labels are dominant. For $k = 0$, we only compute the fastest connection and ignore penalty. We found that k below 10 is a good value that balances preprocessing performance and quality of the solutions at query time. However, it is not essential for precomputation, as in our experience, it reduces the precomputation time and RAM only by a constant factor of about 2.

Perform only one global search. We combine the global arrival and the global departure search (Section 4.2.7) by combining the initial labels. Therefore, for a hub X, the combined global search starts from labels of cost zero at all transfer nodes of station X, and for all nearby stations $A \in \mathcal{N}_X \setminus \{X\}$ from labels of cost walk(X,A) at all transfer nodes of station A. Selecting dominant labels with the arrival chain algorithm regards all labels. Also, we store the transfer patterns in one DAG. The extraction of the transfer patterns needs to distinguish between the labels that originated at station X and those that not, as we need to prepend station X to the transfer pattern of the latter.

Prune at hubs without transfer. In local searches, we mark labels as inactive that just travel through hubs without transfer. This measure reduces the number of access stations without affecting long distance queries too much. However, we observed two problems for which we provide fixes.

The first problem is that we force a transfer at the hub where a transfer is potentially not necessary. We fix this during query graph construction. For every path

$\langle C^{\mathrm{d}}, X^{\mathrm{a}}, X^{\mathrm{d}}, D^{\mathrm{a}} \rangle$ through hub X that represents a transfer at X, we add an edge $(C^{\mathrm{d}}, D^{\mathrm{a}})$ iff there is a direct connection between C and D.

The second problem can arise if the first hub X is on a direct connection from a station A to another hub Y. For example, assume that we have an optimal connection with transfer pattern $AYYB$, and hub X is on the direct connection from A to Y. Also, assume that there is a connection with transfer pattern $XCCB$ that is slightly faster than the subconnection with transfer pattern $XYYB$, both with same penalty score. Then the global search from X would only compute the transfer pattern $XCCB$ and not $XYYB$. Thus the search in the query graph could only find the connection with transfers at X and C, and not the one with just a single transfer at Y. We observed that usually stations X and Y are pretty close. So we reactivate each inactive label at stations close to X (a few hundred meters) that became inactive due to this heuristic and that does not represent a connection that has a transfer after passing through station X. This is a simple postprocessing step after we finished the multi-criteria variant of Dijkstra's algorithm and before we use the arrival chain algorithm. Note however, that we do not compute *all* inactive labels within this distance from X, as we stop generating labels as soon as all unsettled labels are inactive. In theory, we could enforce the computation of these labels by storing the distance since the label became inactive. But we resorted to a more practical and space-efficient solution. We additionally observed that mostly queries where affected where also the hub was close to the source station. So we only stop generating labels if at least a certain number of labels (around a million) is settled.

Drop rare transfer patterns. Rare transfer patterns usually occur at times when services change, for example when service frequencies change in the night. Due to momentary shifts in the time schedule, some transfer pattern is optimal at this certain time of day. The transfer pattern that is optimal at all other times is usually not much worse, and therefore we drop such a rare transfer pattern. Of course, sometimes there are extra connections for rush hours that are significantly faster and we do not want to drop these although they are rare. So we want to ensure that all dropped transfer patterns are *covered* by the remaining transfer patterns with delay and cost increase bounded by a percentage x_d and x_c. More formally, a transfer pattern is covered by a set of transfer patterns iff for each connection P with this transfer pattern there are connections Q and Q' having their transfer patterns in the set with

1. $\mathrm{dep}(P) \leq \mathrm{dep}(Q)$ and $\mathrm{dep}(P) \leq \mathrm{dep}(Q')$,
2. $\mathrm{arr}(Q) - \mathrm{arr}(P) \leq x_d \cdot d(P)$,
3. and $(\mathrm{arr}(Q') - \mathrm{arr}(P)) + (p(Q') - p(P)) \leq x_c \cdot (d(P) + p(P))$.

We use a greedy approach to select the transfer patterns between a pair of stations that we want to keep. The transfer patterns are primarily sorted descending by the number of labels they represent and secondarily ascending by average of the sum of duration

and penalty. In this order we test transfer patterns, drop the covered ones and select the remaining. For efficiency, the coverage test only considers the connections represented by the labels computed by the multi-criteria variant of Dijkstra's algorithm, and not all connections of a transfer pattern.

Single-criterion search in the query graph. Additionally to a multi-criteria search, we also consider a single-criterion search in the query graph that only keeps the label with the smallest duration, with ties broken by lower penalty. We stop the search immediately after the first label is settled at the target station. By ordering the priority queue by the sum of duration and penalty, we do not necessarily compute the earliest arriving connection, but our investigations showed that it is in almost all cases a very good connection from a human prospective. Furthermore, we apply the A^* heuristic to goal-direct the search using minimal durations between station pairs (computed along with the direct-connection data structure) as lower bounds. We compute the potential function for A^* by an initial backward search from the target stations before the actual search [34].

4.2.10 Experiments

Environment. The experimental results we provide in this section are for a fully-fledged C++ implementation. Our experiments were run on a compute cluster of Opteron and Xeon-based 64-bit servers. Queries were answered by a single machine of the cluster, with all data in main memory. Note that we did not have exclusive access to machines of the cluster, so the timings may not be 100% accurate.

Instances. We ran our experiments on three different networks: the train + local transport network of most of Switzerland (PT-CH), the complete transport network of the larger New York area (PT-NY), and the train + local transport network of much of North America (PT-NA). We modeled each network as a time-dependent graph, and Table 4.27 summarizes the different sizes and types.

name	#stations [$\times 10^3$]	#nodes [$\times 10^6$]	#edges [$\times 10^6$]	space [MiB]	type
PT-CH	20.6	3.5	11.9	64	trains + local, well-structured
PT-NY	29.4	16.7	79.8	301	mostly local, poor structure
PT-NA	338.1	113.2	449.1	2 038	trains + local, poor structure

Table 4.27: The three public transportation graphs from our experiments.

Setup. We distinguish between four *precomputation settings* and four *query settings*. They always include transfers with walking and we include all the refinements from Section 4.2.8. Our precomputation setting *without hubs* is based on Section 4.2.5, the setting *with hubs* is based on Section 4.2.7, the 3 legs setting further uses just the 3 legs heuristic of Section 4.2.9, and the *heuristic* setting uses all tricks of Section 4.2.9.

The four query settings result from two types of queries with two different types of searches in the query graph, each. We use station-to-station queries based on Definition 4.18 and location-to-location queries based on Definition 4.30. The first search type performs a time-dependent multi-criteria Dijkstra search on the query graph that uses domination in the Pareto sense, orders the priority queue lexicographically by duration and penalty, and only stops after all unsettled labels are dominated by the labels at the target node. So in case of the first and second precomputation setting, the result is *exact* in the sense of Definition 4.18 or 4.30. The second search type performs only the single-criterion search described in Section 4.2.9.

name	precomp. time [min]	output size [MiB]	query time [μs]
PT-CH	< 1	68	2
PT-NY	4	335	5–9
PT-NA	49	3 399	9–14

Table 4.28: Direct-connection data structure: construction time and size. The query time range is from getting the fastest to all Pareto-optimal connections.

Direct-connection queries. Table 4.28 shows that the preprocessing time for the direct-connection data structure is negligible compared to the transfer patterns precomputation time. The space requirement is from 3 MiB per 1000 stations for PT-CH to 10 MiB per 1000 stations for PT-NY and PT-NA. A query takes from 2 μs for PT-CH to around 10 μs for PT-NY and PT-NA. Note that the larger direct-connection query time for PT-NY and PT-NA is a yardstick for their poor structure (not for their size).

Transfer patterns precomputation. We analyze the effect of hubs and heuristics on the precomputation in Table 4.29. Without hubs, the size of the transfer patterns is more than 18 GiB for PT-CH. With hubs, we significantly reduce the required size to around 800 MiB. But even on this well-structured network, the total precomputation time does not decrease significantly. The local searches consumes 96% of the whole precomputation time, broadly showing the need for an improved local search. Just by employing the 3 legs heuristic, we reduce the time for the local search by a factor of 9. Using additionally the remaining heuristics from Section 4.2.9 does not further reduce the time on PT-CH but reduces it by a factor of 2 for PT-NY. The tightened dominance relation and dropping rare transfer patterns reduces the output size of the global searches

by a factor of 3.1–3.8. The time for global searches is reduced by a factor of 5–6 resulting from a single global search per station and tightened dominance relation.

The heuristic precomputation setting is the only practically feasible one for the large PT-NA instance. We will compare all three instances in this setting. The precomputation time is 20–40 (CPU core) hours per 1 million nodes and the resulting (parts of) transfer patterns can be stored in 10–50 MiB per 1000 stations. These ratios depend mostly on the structure of the network (best for PT-CH, worst for PT-NY and PT-NA), and not on its size.

name		precomp. time [h]		output size [MiB]		#TP/station pair	
		local	global	local	global	local	global
PT-CH	w/o hubs	–	635	–	18 562	–	11.0
PT-CH	w/ hubs	562	24	229	590	2.6	25.8
PT-CH	3 legs	64	24	131	590	2.2	25.8
PT-CH	heuristic	57	4	60	154	2.0	6.8
PT-NY	3 legs	1 359	306	2 311	2 451	5.0	27.0
PT-NY	heuristic	724	64	787	786	3.7	16.4
PT-NA	heuristic	2 632	571	6 849	7 151	3.4	10.5

Table 4.29: Transfer patterns precomputation times and results.

Query graph construction and evaluation. Table 4.30 shows that, on average, query graph construction and evaluation take 5 μs and 15 μs per edge, respectively. The typical number of edges in a query graph for a station-to-station query (1:1) is below 1000 and the typical query time is below 10 ms. Location-to-location queries with 50 source and 50 target stations (50:50) take about 50 ms.

name			constr.	#edges				search	eval.	#edge
			[ms]	50	mean	90	99		[ms]	eval.
PT-CH	w/o hubs	1:1	< 1	32	34	56	86	Pareto	< 1	89
PT-CH	w/ hubs	1:1	1	189	264	569	1 286	Pareto	3	540
PT-CH	heuristic	1:1	< 1	80	102	184	560	Pareto	< 1	194
PT-NY	heuristic	1:1	2	433	741	1 917	3 597	Pareto	6	721
PT-NY	heuristic	1:1	2	433	741	1 917	3 597	single	3	248
PT-NY	heuristic	50:50	32	3 214	6 060	15 878	35 382	single	18	1 413
PT-NA	heuristic	1:1	2	261	536	1 277	3 934	Pareto	10	705
PT-NA	heuristic	1:1	2	261	536	1 277	3 934	single	5	321
PT-NA	heuristic	50:50	22	2 005	3 484	7 240	25 775	single	21	1 596

Table 4.30: Average query graph construction time, size, and evaluation time. The third column also provides the median, 90%-ile and 99%-ile.

4.3 Concluding Remarks

Review. Our algorithm for the scenario with realistic transfer durations is successful because of two main contributions. First of all the station graph model, which has just one node per station, is clearly superior to the time-dependent model, that uses multiple nodes per station for the given scenario. We provide efficient algorithms for the link and minima operation that run in almost linear time. Furthermore, a query in our station graph model is faster than in the time-dependent model, as we need to execute these operations less often. Also all known speed-up techniques that work for the time-dependent model should work for our new model. Most likely, they even work better since the hierarchy of the network is more visible because of the one-to-one mapping of stations to nodes and the absence of parallel edges. The second component is the combination of node contraction and the station graph model. With preprocessing times of a few minutes, we answer time queries in half a millisecond. This algorithm is therefore suitable for applications where small query times are very important and can compensate for our restricted scenario.

Our second algorithm based on transfer patterns is designed for the fully realistic scenario. It decomposes the problem of computing optimal connections into the computation of optimal transfer patterns and direct-connection queries. Although the basic idea is very intuitive, we are the first to exploit it in an efficient algorithm. By precomputing the transfer patterns in advance, we can find all optimal connections between a pair of stations very fast at query time. The most difficult part of the algorithm, and therefore one of our most significant contributions, is a feasible computation of the optimal transfer patterns. As just a hierarchical approach with hubs does not really accelerate the precomputation, we add intelligent heuristics. This reduces the precomputation time by an order of magnitude, and introduces virtually no errors. The resulting algorithm is able to answer location-to-location queries on poor-structured networks with hundreds of thousands of stations within 50 ms, and is used for public transportation routing on Google Maps.

Future Work. Extending the station graph model to a more realistic scenario seems straightforward. But it is an open problem how to efficiently implement link and minima operation when multi-criteria costs are considered. Therefore, it is more promising to improve the transfer patterns approach. A feasible exact algorithm to compute transfer patterns is desirable, as the current one is only sufficiently fast on large networks when heuristics are used. For that, it seems that we need a completely new definition of locality, that allows an effective pruning of local searches. Again, it is an open problem whether this is even possible, as even for road networks, we cannot fully understand the observed efficiency of hierarchical algorithms. From a practical viewpoint, we mainly want to reduce the precomputation time and the query time. To reduce the precomputation time, currently the local searches seem to be the biggest problem. However, for even larger graphs, we can expect the global searches to become more time-consuming.

So it may be good to contract the graph before we execute global searches. To reach the latter goal of reducing the query time, we need to reduce the size of the query graphs, and/or speed up the direct-connection queries. One potential idea would be to add goal-direction for the selection of the access stations. Interestingly, both goals a closely related. We can only run global searches on a contracted graph when we have access stations not only at the source, but also at the target. This further increases the number of transfer patterns in the query graph, so techniques to reduce it are required if we do not want an increasing query time.

The transfer patterns also allow further interesting types of queries. It would be interesting to provide *guidebook routing*, that is a small set of transfer patterns between a pair of stations, that provide at all times almost optimal connections. More concrete, also the answer of *profile queries* is desired, that compute all optimal connections for a whole departure time window.

References. Section 4.1 is based on a technical report [60] and a conference paper [61] solely published by the author of this thesis. Section 4.2 is based on a conference paper [10], which the author of this thesis published together with Hannah Bast, Erik Carlsson, Arno Eigenwillig, Chris Harrelson, Veselin Raychev, and Fabien Viger. Some wordings of these articles are used in this thesis.

5

Flexible Queries in Road Networks

5.1 Central Ideas

A major drawback of most existing speed-up techniques (Section 1.2) is their inflexibility. They are very fast at answering shortest-path queries between a source node s and a target node t, but due to the performed precomputation, they do not allow to specify further query parameters. In this chapter, we consider the scenario with multiple edge weights (Section 5.2) and the scenario with edge restrictions (Section 5.3). We show how to augment the important basic concepts of speed-up techniques (Chapter 3) to these flexible scenarios. Remember our definition of a flexible scenario from Section 2.2.3: For each value of the query parameter p, we can construct a static graph to answer arbitrary shortest path queries. So in principle, we can apply the basic concepts on each of these graphs separately. However, this is not efficient and in practice often not feasible. Working directly on the flexible graph allows to exploit the special properties of the specific flexible scenario under consideration. The augmentation of the basic concepts usually works as follows:

- The *node contraction* adds shortcuts so that the shortest paths between the remaining nodes are preserved *for all* values of the query parameter p. In general, this would require separate witness searches for each value of p, to decide on the necessity of the shortcuts. But for the specific flexible scenarios that are considered in this theses, we are able to reduce the number of witness searches without losing exactness: We can still guarantee that all shortest-path distances are preserved. Also, we do not add too many unnecessary shortcuts. The main difficulty was to find these few efficient witness searches, as they depend on the currently regarded flexible scenario.

- A straightforward adaption of *ALT* would use lower bounds that are valid for all values of the query parameter p. However, as these are usually not very tight for most values of p, it is better to compute several lower bounds for selected values of p. Then, the tightest lower bounds that are still provably lower bounds are used to answer a specific query with its given value of p. Sometimes, it is even possible to

combine lower bounds to get a lower bound for a value of p that was not selected for precomputation. However, this again depends on the specific flexible scenario.

Adding flexibility has its price, usually the precomputation time and space, and the query time increase compared to the static scenario (Section 2.2.1). Nevertheless, the resulting algorithms have a significant speed-up over Dijkstra's algorithm. Still, it is desirable to develop new techniques that are only possible in specific flexible scenarios. We came up with the following enhancing concepts concerning *node contraction*:

- Use not only a single *node order* for node contraction, but several depending on the value of the query parameter p. The idea behind this concept is that the importance of nodes changes, depending on the value of p, and it is beneficial for query performance to adapt the node order. Often unimportant nodes, for example within a living area, where all streets are the same, stay unimportant independent of the value of p. So a refinement of this concept performs an initial contraction and adapts node orders only for a smaller core. This reduces preprocessing time and space, as computing different node orders effectively requires separated computations.

- We ensure that for each value of the query parameter p, there are sufficient shortcuts to preserve shortest-path distances. The witness searches during node contraction also ensure that there are not too many unnecessary shortcuts. Still, for a specific value of p, we usually do not need all of them. So we should store some additional information to detect at query time the necessary shortcuts for the current value of p. Only relaxing the necessary shortcuts, and pruning the other ones, speeds up the query.

5.2 Multiple Edge Weights

The flexible scenario with multiple edge weight functions considers a graph $G = (V,E)$ with multiple edge weight functions $c^{(1)}(e),\dots,c^{(r)}(e)$ as introduced in Section 2.2.3. The query parameter is an r-dimensional coefficient vector used to linearly combine the edge weights. We restrict ourselves to exactly two edge weight functions $c^{(1)}(e)$ and $c^{(2)}(e)$. We combine them using a parameter $p \in [L,U] := \{x \in \mathbb{Z} \mid L \leq x \leq U\}$ to a single edge weight $c_p(e) := c^{(1)}(e) + p \cdot c^{(2)}(e)$. It is necessary that $c_p(e)$ is non-negative for all $p \in [L,U]$. Having a discrete parameter in a bounded finite interval allows the development of an very efficient algorithm that exploits each of these properties. Our goal is to efficiently answer a flexible query following Definition 5.1.

Definition 5.1 *A* flexible query *in the scenario with multiple edge weights computes the shortest-path distance $\mu_p(s,t)$ between a source node s and a target node t for a query parameter p subject to the edge weight function $c_p(e) := c^{(1)}(e) + p \cdot c^{(2)}(e)$.*

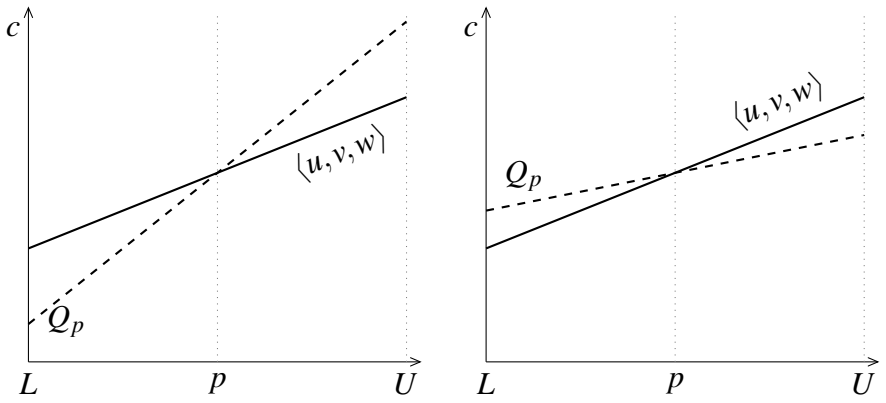
Note that for simplicity we assume that p is integral. However, we will see that it is only important that p is discrete, that is given a value of p, we can compute the next $(+1)$ and the previous (-1) value. An equivalent combination of the two edge weight functions would be $c_p(e) := (1-p) \cdot c^{(1)}(e) + p \cdot c^{(2)}(e)$. The equivalence is easily visible by the equation $(1-p) \cdot c^{(1)}(e) + p \cdot c^{(2)}(e) = c^{(1)}(e) + p \cdot \left(c^{(2)}(e) - c^{(1)}(e)\right)$.

5.2.1 Node Contraction

Remember from Section 3.2, that to contract a node $v \in V$, we remove it and all its adjacent edges from the graph and add shortcuts to preserve shortest path distances in the remaining graph. All original edges together with all shortcuts are the result of the preprocessing. As we want to add no unnecessary shortcuts, we face the following many-to-many shortest path problem: For each uncontracted source node $u \in V$ with $(u,v) \in E$, each uncontracted target node $w \in V$ with $(v,w) \in E$ and each integer parameter $p \in [L,U]$, we want to compare the shortest u-w-paths Q_p with minimal $c_p(Q_p)$ with the shortcut $\langle u,v,w \rangle$ in order to decide whether the shortcut is really needed.

Compared to single-criteria contraction, we cannot avoid parallel edges. However, in practice, their number is very small. Furthermore, their number is bounded by the number of possible values of p, and also by the number of Pareto-optimal paths. So we are never worse than in a Pareto-optimal setting. We keep identifying edges by their two endpoints since the particular edge should always be clear from the context.

Witness Search. We extend the concept of a *witness* introduced in Section 3.2 to our flexible scenario. Here, a witness allows to omit a shortcut for a single value of p. A simple implementation of the *witness search* could perform for each value of p a forward shortest-path search in the remaining graph from each source, ignoring node v,



(a) $c^{(2)}(Q_p) \geq c^{(2)}(u,v) + c^{(2)}(v,w)$ (b) $c^{(2)}(Q_p) \leq c^{(2)}(u,v) + c^{(2)}(v,w)$

Figure 5.1: The cost of a path represents a linear function in dependence of the parameter value p.

until all targets have been settled. Still, if the cardinality of $[L,U]$ is large, this is infeasible in practice as we need to perform too many witness searches. Another possible implementation could use a Pareto-Dijkstra, cf. Section 1.3.3. Using a Pareto-Dijkstra would also lift the requirement to have discrete parameter values. However, this requires the storage of multiple labels per node and also the results of Delling and Wagner [45] suggest that there can be too many labels. So instead, we do something tailored to our linear combination of two edge weights combined with discrete parameter values: Let Q_p be a u-w-path with $c_p(Q_p) \leq c_p(u,v) + c_p(v,w)$. As depicted in Figure 5.1, $c^{(2)}(Q_p)$ can be seen as slope of a linear function. Therefore, we observe that if $c^{(2)}(Q_p) \geq c^{(2)}(u,v) + c^{(2)}(v,w)$, then $c_q(Q_p) \leq c_q(u,v) + c_q(v,w)$ for all $q \in [L,p]$. And if $c^{(2)}(Q_p) \leq c^{(2)}(u,v) + c^{(2)}(v,w)$, then $c_q(Q_p) \leq c_q(u,v) + c_q(v,w)$ for all $q \in [p,U]$. This observation implies Lemma 5.2.

Lemma 5.2 *Let $\langle u,v,w \rangle$ be a potential shortcut. Any parameter interval $[L,U]$ can be partitioned into three, possibly empty partitions $[L,L']$, $[L'+1,U'-1]$ and $[U',U]$ with integers L',U' and the following properties:*

(1) If $[L,L']$ is not empty, then there exists a single witness path for all $p \in [L,L']$.

(2) If $[U',U]$ is not empty, then there exists a single witness path for all $p \in [U',U]$.

(3) If $[L'+1,U'-1]$ is not empty, then for all values of p in it, there exists no witness path.

Our witness search (Algorithm 5.1) will compute the interval $[L'+1,U'-1]$ from Lemma 5.2, using the previous observation. First, we start a parameter increasing witness search (Algorithm 5.2) with $p := L$. When we find a witness path Q, we compute

the largest $p' \geq p$ for which Q is still a witness path. This works in constant time since the cost of a path is basically a linear function over the parameter p, and p' is the possible intersection of the cost of the witness path and the cost of the possible shortcut. Then, we continue with $p := p' + 1$ and repeat the procedure increasing p until we either reach U or find no witness path. Note that because of this '+1', we only support discrete parameter values. If we reach U, we know that no shortcut is necessary as $[L'+1, U'-1]$ is empty, and our witness search is done. Otherwise, we perform the symmetric parameter increasing witness search, starting with $p := U$ to compute U'.

Algorithm 5.1: WitnessSearch($\langle u, v, w \rangle$, L, U)

input : path $\langle u, v, w \rangle$, interval $[L, U]$
output : interval $[L'+1, U'-1]$ for which a shortcut "could be" necessary

```
1 L' := ParamIncreasingWitnessSearch(⟨u,v,w⟩, [L,U]);
  // no witness necessary
2 if L' ≥ U return ∅;
  // else [L'+1, U'-1] ≠ ∅
3 U' := ParamDecreasingWitnessSearch(⟨u,v,w⟩, [L,U]);
4 return [L'+1, U'-1];
```

Algorithm 5.2: ParameterIncreasingWitnessSearch($\langle u, v, w \rangle$, L, U)

input : path $\langle u, v, w \rangle$, interval $[L, U]$
output : p with witness on $[L, p]$

```
1 p := L;
2 while p smaller or equal to U do
      // Witness search for p returning a potential witness P.
3     P := WitnessSearch(u, w, v, p);
4     if c_p(P) > c_p(⟨u,v,w⟩) return p − 1;
5     p := max{k ∈ ℤ | c_k(P) ≤ c_k(⟨u,v,w⟩)} + 1;
6 return U;
```

The *necessity interval* $[L'+1, U'-1]$ obtained by the witness search is stored as $NI(e)$ with a necessary shortcut e to remember that e is only necessary for $p \in NI(e)$. We will use $NI(e)$ for our query algorithm that we will describe later. But we can also use the necessity intervals to reduce the number of witness searches during the contraction of a node v, see Algorithm 5.3. Given an incoming edge (u, v) and an outgoing edge (v, w), the potential shortcut (u, w) representing the path $\langle u, v, w \rangle$ is only necessary for the intersection of the necessity intervals of (u, v) and (v, w).

As our witness searches may be limited, for example by the number of hops, we cannot guarantee that we compute the smallest interval. But we can guarantee that we

Algorithm 5.3: Contraction(v, L, U)

input : node v, interval $[L,U]$

foreach $(u,v) \in E$ *with u not contracted,* $(v,w) \in E$ *with w not contracted* **do**
  // Shortcut only potentially necessary for intersection of necessity intervals.
  $I := NI(u,v) \cap NI(v,w) \cap [L,U]$;
  if $I = \emptyset$ **then continue**;
  $[L'+1, U'-1]$:=WitnessSearch($\langle u,v,w \rangle$, I);
  if $[L'+1, U'-1] \neq \emptyset$ **then**
    // Add shortcut (u,w) with necessity interval $[L'+1, U'-1]$.
    $e := (u,w)$;
    $E := E \cup \{e\}$;
    $c^{(1)}(e) := c^{(1)}(u,v) + c^{(1)}(v,w)$;
    $c^{(2)}(e) := c^{(2)}(u,v) + c^{(2)}(v,w)$;
    $NI(e) := [L'+1, U'-1]$;

always compute an interval containing the smallest. The correctness of the contraction as stated in Lemma 5.3 follows directly.

Lemma 5.3 *Consider the contraction of node v. Let* $(u,v) \in E$, $(v,w) \in E$ *with u, w not being contracted. A shortcut with the cost of* $\langle u,v,w \rangle$ *is only omitted if for each* $p \in [L,U]$ *there exists a witness path* P_p *with* $c_p(P_p) \leq c_p(\langle u,v,w \rangle)$. *The necessity interval of a shortcut includes all values of p where no witness path exists.*

In practice, we observe an average of two performed shortest path queries. Since in some cases a large number of queries may be necessary to find the minimal necessity interval, we limit the number of single source shortest paths queries to 30.

Parameter Interval Reduction. Due to the locality of our witness searches, that for example uses a hop limit, we may insert unnecessary shortcuts. Also edges of the original graph might not be necessary for all parameters. So whenever we perform a witness search from u, we compare the incident edges (u,x) with the computed path $\langle u,\ldots,x \rangle$. When there are values of p where $\langle u,\ldots,x \rangle$ is shorter than (u,x), we reduce its necessity interval and delete the edge when the interval is empty.

Parameter Splitting. Preliminary experiments revealed that a single node order for a large parameter interval will result in too many shortcuts. That is because a single node order is no longer sufficient when the shortest paths significantly change over the whole parameter interval. One simple solution would be to split the intervals into small adequate pieces beforehand, and contract the graph for each interval separately. But we can do better: we observed that a lot of shortcuts are required for the whole parameter

interval. Such a shortcut for a path $\langle u, v, w \rangle$ would be present in each of the constructed hierarchies that contracts v before u and w. Therefore, we use a classical divide and conquer approach. We repeatedly split the parameter intervals during the contraction and compute separated node orders for the remaining nodes. For that, we need to decide on when to split and how to split the interval.

A split should happen when there are too many "differences" in the classification of importance between different values of p in the remaining graph. An indicator for these "differences" are the necessity intervals of the added shortcuts. When a lot of *partial shortcuts*, i. e. shortcuts not necessary for the whole parameter interval, are added, a split seems advisable. So we trigger the split when the number of partial shortcuts exceeds a certain limit. However, in our experience, this heuristic needs to be adjusted depending on the metrics used. One reason for this imperfection is the difficult prediction of the future contraction behavior. Sometimes it is good to split earlier although the contraction currently works well and not too many partial shortcuts are created. But due to the shared node order, the contraction becomes more difficult later, even when we split then.

A very simple method to split a parameter interval is to cut into halves (*half split*). However, this may not be the best method in every case. The "different" parameters can be unequally distributed among the parameter interval and a half split would return two unequally difficult intervals. So we may also try a *variable split* where we look again on the shortcuts and their necessity intervals to improve the parameter interval splitting. One possibility is to split at the smallest (largest) parameter which lets more than half of the edges become necessary. If no such parameter exists, we cut into halves.

To reduce main memory consumption, we also use hard disk space during contraction. Before we split, we swap the shortcuts introduced since the last split to disk. When we have finished contraction of one half of an interval, and we need to contract the other half, we load the state of the graph before the split from disk. This saves us a significant amount of main memory and allows the processing of large graphs.

We limit the witness search to initially 8 hops, cf. Section 3.2.1. This may add superfluous shortcuts but does not affect the correctness. Furthermore, as we have to contract dense cores more often due to the splitting, we use a staged hop limit [67] to reduce the precomputation time: Once the average degree in the remaining graph reaches 30 we switch to a hop limit of 6.

Node Ordering. We select the node order as described in Section 3.2 using a heuristic that keeps the nodes in a priority queue, sorted by some estimate of how attractive it is to contract a node. We measure the attractiveness with a linear combination of several priority terms. After some initial tests with previously introduced priority terms [67], we decided to use the following terms. The first term is a slightly modified version of the *edge difference*; instead of the difference, we count the number of *added shortcuts* (factor 15) and the number of *deleted edges* (factor -4). The second term is for uniformity, namely *deleted neighbors* (factor 15). The third term favors the contraction in more sparse regions of the graph. We count the number of relaxed edges during a witness

search as the *search space* (factor 8). Our last terms focus on the shortcuts we create. For each shortcut we store the number of original edges it represents. Furthermore, for every new shortcut (u,w) the contraction of a node v would yield, we calculate the difference between the number of original edges represented by (u,v) and (u,w) (both factor 1). Also, we use the same heuristics for priority updates, i. e. updating only neighbors of a contracted node and using *lazy updates*.

5.2.2 A* Search using Landmarks (ALT)

For a hierarchy given by some node order, we call the K most important nodes the *core* of the hierarchy as defined in Section 3.4. We observed that preprocessing the core takes long because the remaining graph is dense and we have to do it several times due to parameter splitting. Therefore we use the combination of node contraction and ALT [69] described in Section 3.4. Remember that we have a choice to use ALT on an uncontracted or a contracted core. Not contracting the core speeds up the preprocessing, whereas contracting it speeds up the query. For both variants, we extended ALT to our bi-criteria scenario. We need to compute appropriate landmarks and distances for that.

Again, the straightforward approach, to compute landmarks and distances for every value of p separately, is too time-consuming. Also, space consumption becomes an issue if we compute a landmark set for every value of p. Another idea would be to compute lower bounds separately for each edge weight function, and then combine them given the current value of p. But we can do better: Given a core for a parameter interval $[L,U]$, we compute two sets of landmarks and distances: one for the edge weight function c_L and one for c_U.

Given two nodes s and t, a lower bound $\phi_L \leq \mu_L(s,t)$ for parameter value L and a lower bound $\phi_U \leq \mu_U(s,t)$ for parameter value U can be combined to a lower bound $\phi_p \leq \mu_p(s,t)$ for any parameter value $p \in [L,U]$:

$$\phi_p := (1-\alpha)\cdot\phi_L + \alpha\cdot\phi_U \text{ with } \alpha := (p-L)/(U-L) \tag{5.1}$$

The correctness (Lemma 5.5) of the lower bound (5.1) is mainly based on Lemma 5.4.

Lemma 5.4 *Let the source node s and target node t of a query be fixed. Let $\phi(p) := \mu_p(s,t)$ be the shortest path distance for a real-valued parameter p in $[L,U]$. Then, $\phi(\cdot)$ is* concave*, i. e. for $p,q \in [L,U], \alpha \in [0,1] : \phi((1-\alpha)p+\alpha q) \geq (1-\alpha)\phi(p)+\alpha\phi(q)$.*

Proof. Assume p,q,α as defined above exist with $\phi((1-\alpha)p+\alpha q) < (1-\alpha)\phi(p) + \alpha\phi(q)$. Let P be a shortest path for parameter $(1-\alpha)p+\alpha q$. In the case $c_p(P) \geq \phi(p)$, we deduce directly $c_q(P) < \phi(q)$, and analogously for $c_q(P) \geq \phi(p)$, we deduce $c_p(P) < \phi(p)$. This contradicts the definition of $\phi(\cdot)$. □

Lemma 5.5 *The lower bound ϕ_p in equation (5.1) is a feasible lower bound.*

Proof. This is a direct consequence of Lemma 5.4 and the choice of α, as it holds $(1-\alpha)c_L + \alpha c_U = c_p$. □

The lower bound (5.1) is lower in quality as an individual bound for any value of p, since shortest paths are not all the same for $p \in [L,U]$, but is better than the lower bound obtained from landmarks for the two input edge weight functions $c^{(1)}(\cdot)$ and $c^{(2)}(\cdot)$.

5.2.3 Query

Dijkstra's algorithm (Section 3.1) can be easily adapted to the flexible scenario with multiple edge weights by computing $c_p(e)$ for each relaxed edge e on demand. The same adaption is done to the original CH query algorithm (Section 3.2.2). Furthermore, we use the necessity intervals to only relax edges that are necessary for the current value of p. The parameter splitting during the contraction results in multiple node orders. Therefore we have to augment the definition of the upward and downward graph. Our upward/downward graph contains all edges that are directed upwards/downwards in the hierarchy for their parameter interval. Therefore, the upward edges are

$$\overrightarrow{E} := \{e = (u,v) \in E \mid \exists p : p \in NI(e), u \text{ contracted before } v \text{ for } p\}$$

and the downward edges are

$$\overleftarrow{E} := \{e = (u,v) \in E \mid \exists p : p \in NI(e), u \text{ contracted after } v \text{ for } p\} \quad .$$

If there are edges that are upwards and downwards, depending on a parameter in their necessity interval, we split them into several parallel edges and adjust their necessity interval. Otherwise the definition of the search graph G^* in Section 3.2.2 remains the same. Then, our query Algorithm 5.4 can recognize an edge not being directed upwards/downwards for the given value of p by looking at its necessity interval. Compared to the original Algorithm 3.3 in Section 3.2.2, we only changed Lines 15 and 16.
During experiments, we observed that a query scans over a lot of edges that are not necessary for the respective parameter value. We essentially scan much more edges than we relax. We alleviate this problem with buckets for smaller parameter intervals. As data structure, we use an adjacency array, with a node array and an edge array. As an additional level of indirection, we add a bucket array, storing the edge indices necessary for each bucket. These indices are ordered like the edge array, so a single offset into this additional array is enough per node. Each node stores one offset per bucket array. By this method we essentially trade fast edge access for space consumption. A lot of edges are necessary for the whole parameter interval, almost all edges of the original graph and additionally many shortcuts. We store them separately, since otherwise, we would have an entry in each bucket array for each of these edges. This single action makes the buckets twice more space-efficient. Note that we can access the edges resident in all buckets without the further level of indirection, we just use another offset per node that points into a separate array.

Algorithm 5.4: FlexibleEdgeWeightCHQuery(s, t, p)

input : source s, target p, parameter value p
output : shortest path distance δ

1 $\overrightarrow{\delta} := \langle \infty, \ldots, \infty \rangle$; // tentative forward distances
2 $\overleftarrow{\delta} := \langle \infty, \ldots, \infty \rangle$; // tentative backward distances
3 $\overrightarrow{\delta}(s) := 0$; // forward search starts at node s
4 $\overleftarrow{\delta}(t) := 0$; // backward search starts at node t
5 $\delta := \infty$; // tentative shortest path distance
6 $\overrightarrow{Q}$.insert(0, s); // forward priority queue
7 $\overleftarrow{Q}$.insert(0, t); // backward priority queue
8 $\sim := \rightarrow$; // current direction
9 **while** $(\overrightarrow{Q} \neq \emptyset)$ **or** $(\overleftarrow{Q} \neq \emptyset)$ **do**
10 **if** $\delta < \min\left\{\overrightarrow{Q}.\text{min}(), \overleftarrow{Q}.\text{min}()\right\}$ **then break**;
11 **if** $\widetilde{Q} \neq \emptyset$ **then** $\sim := \neg \sim$; // interleave direction, $\neg \leftarrow = \rightarrow$ and $\neg \rightarrow = \leftarrow$
12 $(\cdot, u) := \widetilde{Q}$.deleteMin(); // u is settled
13 $\delta := \min\left\{\delta, \overrightarrow{\delta}(u) + \overleftarrow{\delta}(u)\right\}$; // u is potential candidate
14 **foreach** $e = (u, v) \in E^*$ **with** $\sim(e)$ **do** // relax edges
15 **if** $p \in NI(e)$ **and** $\left(\widetilde{\delta}(u) + c_p(e)\right) < \widetilde{\delta}(v)$ **then** // shorter path via u?
16 $\widetilde{\delta}(v) := \widetilde{\delta}(u) + c_p(e)$; // update tentative distance
17 $\widetilde{Q}$.update$\left(\widetilde{\delta}(v), v\right)$; // update priority queue
18 **return** δ;

Theorem 5.6 *Given a source node s, a target node t and a parameter $p \in [L,U]$, our CH query for the flexible scenario with two edge weights computes $\mu_p(s,t)$.*

Proof. Let us construct a static graph induced by the edges $\{e \in E \mid p \in NI(e)\}$ with edge weight function $c_p(e)$. Obviously, our query for parameter value p corresponds to the static CH query on this constructed graph. As Lemma 5.3 proves that we add all necessary shortcuts for p, we can deduce from Corollary 3.2 that our algorithm computes $\mu_p(s,t)$. □

Combination with ALT. We perform the two-phased query of Section 3.4, but augmented to our flexible scenario. We use the augmented CH query algorithm of the previous paragraph, and perform the potential computation as described in Section 5.2.2.

We also looked into storing proxy nodes in advance instead of computing them at query time. This would result in an extra number of $3 \cdot |\#cores|$ integers we would have to store for every node. And the speedup of the query would be marginal, as the proxy search is very fast and takes only a small part of the whole query time. We observed a number of 7% to 13% of the settled nodes to be a result of the proxy searches. For the

relaxed edges, the part of the proxy search varied between 3% and 5%. So we decided to compute the proxy nodes for every query anew.

As the potential functions obtained by the landmarks are not necessarily consistent, we have a choice of making them consistent for queries on an uncontracted core, cf. Section 3.4. However, this did not pay off in preliminary experiments.

Theorem 5.7 *Given a source node s, a target node t and a parameter $p \in [L,U]$, our query for the flexible scenario with two edge weights combined with ALT computes $\mu_p(s,t)$.*

Proof. Let us construct a static graph G_p induced by the edges $\{e \in E \mid p \in NI(e)\}$ with edge weight function $c_p(e)$. Obviously, our query for parameter value p corresponds to a static query on this graph. As Lemma 5.3 proves that we add all necessary shortcuts for p, and Lemma 5.5 proves that our lower bounds are feasible, the correctness of our query follows directly from the correctness of the static query [38, Theorem 4.1]. □

Profile Query. Our algorithm can also be utilized to find a set of shortest paths between a given source and target node so that for each value of p a shortest path is in the set (Definition 5.8).

Definition 5.8 *A* profile query *between a source node s and a target node t for an interval $[p_1,p_2]$ returns a set $\mathscr{S}$ of s-t-paths such that*

(1) $\forall p \in [p_1,p_2] : \exists P \in \mathscr{S} : c_p(P) = \mu_p(s,t)$

(2) $\forall P \in \mathscr{S} : \exists p \in [p_1,p_2] : c_p(P) = \mu_p(s,t)$

(3) $\forall P,Q \in \mathscr{S} : P \neq Q \Rightarrow \left(c^{(1)}(P) \neq c^{(1)}(Q) \vee c^{(2)}(P) \neq c^{(2)}(Q)\right)$

(1) ensures that there is a shortest path for every value of p, (2) ensures that each path in $\mathscr{S}$ is a shortest path, and (3) ensures that no two paths have same cost. Note that such a set $\mathscr{S}$ is not necessarily unique, as even for a fixed value of p there can be multiple shortest paths. We give an algorithm to compute such a set $|\mathscr{S}|$ with $3 \cdot |\mathscr{S}| - 2$ flexible queries. It is based on the following Lemma 5.9.

Lemma 5.9 (Cutting Point Lemma) *Let P_1 be a shortest s-t-path for a parameter value p_1, P_2 be a shortest s-t-path for a parameter value p_2, and $\left(c^{(1)}(P_1),c^{(2)}(P_1)\right) \neq \left(c^{(1)}(P_2),c^{(2)}(P_2)\right)$. If a shortest s-t-path P' for a parameter value p' exists with $p_1 < p' < p_2$ and $\left(c^{(1)}(P_1),c^{(2)}(P_1)\right) \neq \left(c^{(1)}(P'),c^{(2)}(P')\right) \neq \left(c^{(1)}(P_2),c^{(2)}(P_2)\right)$, then there exists also such a path P at the "cutting point" parameter value p defined by $c_p(P_1) = c_p(P_2)$ as in Figure 5.2.*

Proof. Let us assume that a path P' and parameter value p' exist with $p_1 < p' < p_2$ and $c_{p'}(P') < c_{p'}(P_1)$ and $c_{p'}(P') < c_{p'}(P_2)$. Furthermore, we assume that at p with $c_p(P_1) = c_p(P_2)$ no path P exists that fulfills $c_p(P) < c_p(P_1) = c_p(P_2)$. Also let WLOG $p_1 < p' < p$. Since $c_{p_1}(P_1)$ is minimal over all existing s-t-paths at p_1, $c_{p_1}(P') \geq c_{p_1}(P_1)$. With the assumption also $c_p(P') \geq c_p(P_1)$ holds. This cannot be true for linear functions of degree one, as $c_{p'}(P') < c_{p'}(P_1)$. □

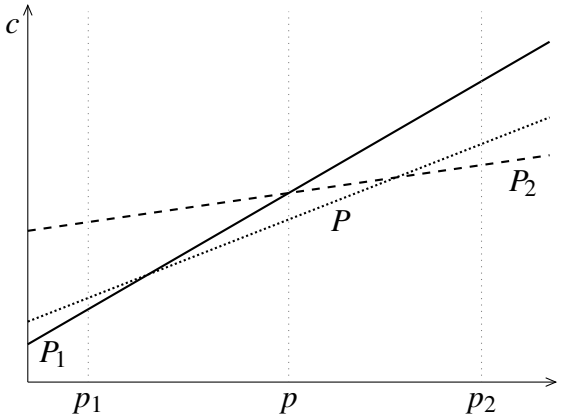


Figure 5.2: The cost of each path is plotted in dependency of the parameter value. The profile query can locate an additional shortest path P in the interval (p_1, p_2) by looking at the "cutting point" p of the shortest paths P_1 and P_2.

Due to Lemma 5.9, we can compute the set $\mathscr{S}$ recursively by a divide and conquer algorithm that divides at the cutting point p, or rather $\lfloor p \rfloor$ or $\lceil p \rceil$ if p is not integral. To decide whether we can stop the recursion, and to add paths to $\mathscr{S}$, we perform flexible queries for $\lfloor p \rfloor$ and $\lceil p \rceil$ to see whether the found paths are better than the ones for p_1 and p_2 (Algorithm 5.5). Note that a second query on $\lceil p \rceil$ is only necessary if the path found by the query for $\lfloor p \rfloor$ is not shorter compared to the path found for p_1. Let $k = |\mathscr{S}|$. For the case $k = 1$ we obviously need two queries. For $k \geq 2$, Lemma 5.10 proves that we need at most $3k-2$ flexible queries. Note that for a continuous interval of parameter values, only $2k-1$ flexible queries would be necessary, as we would not need to round the cutting point.

Lemma 5.10 *Let source node s and target node t be fixed. Let $[p_1, p_2]$ be an integral parameter interval. Let $\mathscr{S}$ be the set of paths computed by our profile query. Let $k := |\mathscr{S}|$. If $k \geq 2$, our profile query needs to perform at most $3k-2$ parameter queries to compute $\mathscr{S}$. Also, this set fulfills the properties of Definition 5.8.*

Proof. Let $k \geq 2$. Let $T(k)$ be the maximum number of flexible queries our profile query needs to perform for an arbitrary integral interval $[p_1, p_2]$, when the flexible queries for parameter values p_1 and p_2 are already performed. We will prove by induction, that $T(k) \leq 3k-4$. This will prove our claim, as this just excludes the two flexible queries

Algorithm 5.5: ProfileSearch(s, t, p_1, p_2, P_1, P_2)

input : source s, target t, integral interval $[p_1, p_2]$, shortest s-t-path P_1 for p_1, P_2 for p_2
output : set $\mathscr{S}$ by Definition 5.8

```
1  determine the "cutting point" p between P_1 and P_2 following Lemma 5.9;
2  perform flexible query for parameter value ⌊p⌋ resulting in path P;
3  if c_⌊p⌋(P) < c_⌊p⌋(P_1) then                          // more than one path in [p_1, ⌊p⌋]
4      S_1 := ProfileSearch(s, t, p_1, ⌊p⌋, P_1, P);
5      if c_⌊p⌋(P) < c_⌊p⌋(P_2) then                      // more than one path in [⌊p⌋, p_2]
6          S_2 := ProfileSearch(s, t, ⌊p⌋, p_2, P, P_2);
7      else S_2 := {P_2};
8  else
9      S_1 := {P_1};
10     perform flexible query for parameter value ⌈p⌉ resulting in path P';
11     if c_⌈p⌉(P') < c_⌈p⌉(P_2) then                     // more than one path in [⌈p⌉, p_2]
12         S_2 := ProfileSearch(s, t, ⌈p⌉, p_2, P', P_2);
13     else S_2 := {P_2};
14 return S_1 ∪ S_2;
```

for the boundary values. Let P_1 be the path found by our query for parameter value p_1 and P_2 for p_2. Let p be the "cutting point": $c_p(P_1) = c_p(P_2)$. For the base case $k = 2$, we need 2 flexible queries, one for $\lfloor p \rfloor$ and one for $\lceil p \rceil$. Induction step: For $k \geq 3$, let P be the path computed for parameter value $\lfloor p \rfloor$. It holds

$$T(k) \leq \begin{cases} 2 + T(k-1) & \text{if } c_{\lfloor p \rfloor}(P) = c_{\lfloor p \rfloor}(P_1) \text{ or } c_{\lfloor p \rfloor}(P) = c_{\lfloor p \rfloor}(P_2) \\ 1 + T(k_1) + T(k_2) & \text{else, with } k_1 = |\mathscr{S}_1|, \text{ and } k_2 = |\mathscr{S}_2| \end{cases}$$

In the first case holds

$$T(k) \leq 2 + T(k-1) \overset{\text{IH}}{\leq} 2 + 3(k-1) - 4 = 3k - 5 < 3k - 4 \ .$$

In the second case we know that the costs of the path P is different from the paths P_1 and P_2, therefore $k_1 \geq 2$ and $k_2 \geq 2$ and we can apply the induction hypothesis. Also $k_1 + k_2 = k + 1$, as the intersection of $\mathscr{S}_1$ and $\mathscr{S}_2$ contains exactly the path P. Therefore holds

$$T(k) \leq 1 + T(k_1) + T(k_2) \overset{\text{IH}}{\leq} 1 + (3k_1 - 4) + (3k_2 - 4) = 3(k+1) - 7 = 3k - 4 \ .$$

What is left is to prove is that for the computed set the properties (1) – (3) of Definition 5.8 hold. Property (2) holds by construction, as we only add a path to $\mathscr{S}$ that is returned by a flexible query. Property (1) holds, as in case that for a s-t-path Q and an interval $[q_1, q_2]$ holds $c_{q_1}(Q) = \mu_{q_1}(s,t)$ and $c_{q_2}(Q) = \mu_{q_2}(s,t)$, then the path Q is a shortest path for the whole interval (Lemma 5.4). Property (3) holds, as we only add paths that have different cost than the boundary paths P_1 and P_2. □

Profile Query with Sampling. As an alternative to a complete profile query, our algorithm can also be used to perform queries for a sample subset $S = \{p_1, p_2, \ldots, p_k\} \subseteq [L,U]$. To do so, we adapt our algorithm from the profile query. We start with a query for the minimal parameter value p_1 and the maximal parameter value p_k in S. For two paths P_i at parameter p_i and P_j at parameter p_j with $p_i < p_j$ we calculate the next query parameter as p_ℓ, $\ell = \lfloor i + (j-i)/2 \rfloor$. By this method we recursively continue. If we find the already known path P_i (P_j) again at p_ℓ, we do not need to continue the recursion between p_ℓ and p_i (p_j).

Approximated Profile Query. Since the large number of queries result in a relatively long computation time for a profile query, we also offer the possibility of an approximated profile query with ε-guarantee:

Lemma 5.11 *For two shortest s-t-paths P_1 at parameter value p_1 and P_2 at parameter value p_2, $p_1 < p_2$ and $c^{(1)}(P_2) \leq (1+\varepsilon) \cdot c^{(1)}(P_1)$ or $c^{(2)}(P_1) \leq (1+\varepsilon) \cdot c^{(2)}(P_2)$ holds: if a shortest s-t-path P at parameter value p exists with $p_1 < p < p_2$ then either $c^{(1)}(P) \leq (1+\varepsilon) \cdot c^{(1)}(P_1) \wedge c^{(2)}(P) \leq (1+\varepsilon) \cdot c^{(2)}(P_1)$ or $c^{(1)}(P) \leq (1+\varepsilon) \cdot c^{(1)}(P_2) \wedge c^{(2)}(P) \leq (1+\varepsilon) \cdot c^{(2)}(P_2)$ holds.*

Proof. First let $c^{(1)}(P_2) \leq (1+\varepsilon) \cdot c^{(1)}(P_1)$. In this case we can use P_2 to approximate P. From $p < p_2$ follows $c^{(2)}(P_2) \leq c^{(2)}(P)$. From $p_1 < p$ follows $c^{(1)}(P) \geq c^{(1)}(P_1)$ and with $c^{(1)}(P_2) \leq (1+\varepsilon) \cdot c^{(1)}(P_1)$ follows $c^{(1)}(P_2) \leq (1+\varepsilon) \cdot c^{(1)}(P)$. In the same way we can use P_1 in case that $c^{(2)}(P_1) \leq (1+\varepsilon) \cdot c^{(2)}(P_2)$ holds. □

Therefore, we can guarantee that for every omitted path P a path P' will be found with $c^{(1)}(P') \leq (1+\varepsilon) \cdot c^{(1)}(P)$ and $c^{(2)}(P') \leq (1+\varepsilon) \cdot c^{(2)}(P)$. Since we essentially need two queries to terminate our search for further paths between two already found paths, the approximation might help to reduce the number of "unnecessary" searches without omitting too many paths. Note that the approximation method can also be applied to our profile query with sampling.

5.2.4 Experiments

Instances. We present experiments performed on road networks from the year 2006, provided by PTV AG. The German road network (GER) consists of 4 692 751 nodes and 10 806 191 directed edges. The European road network (EUR) consists of 29 764 455 nodes and 67 657 778 directed edges. For comparison with Pareto-SHARC, we also performed experiments with their older network of Western Europe (WEU) having 18 017 748 nodes and 42 189 056 directed edges.

Environment. We did experiments on one core of a single AMD Opteron Processor 270 clocked at 2.0 GHz, with 8 GiB main memory and 2×1 MiB L2 cache. Only the preprocessing of EUR has been done on one core of a Intel Xeon 5345 processor clocked at 2.33 GHz with 16 GiB main memory and 2×4 MiB L2 cache, as more main memory was required. We run SuSE Linux 11.1 (kernel 2.6.27) and use the GNU C++ compiler 4.3.2 with optimization level 3.

Annotation. In our tables we denote with *param* the number of different parameters in the interval $[0, x-1]$. For core approaches we may give the kind of core used by a string of two signifiers *core size/#landmarks*. The core size indicates how many nodes the core contains. The second number gives the number of landmarks used in the core. A core is usually uncontracted, only the contracted cores are additionally marked with *C*. The *preproc*essing time is given in the format hh:mm and is split into the *compute* time and the *I/O* time needed for reading/writing to disk. *Query* performance is given in milliseconds. For the *speed-up* we compare our algorithm to the timings of a plain unidirectional Dijkstra algorithm.

Weight Functions. For our experiments, we combined the travel time and the approximate monetary cost for traversing an edge. The travel time was computed from the length of an edge and the provided average speed. To approximate the cost, we chose to calculate the needed mechanical work for a standard car. We use the standard formulas for rolling and air resistance to compute the force $F(v) = F_N \cdot c_r + A \cdot c_w \cdot \frac{\rho}{2} \cdot v^2$ with F_N the normal force, c_r the rolling resistance coefficient, A representing the reference area in square meters, c_w being the drag coefficient, ρ the air density and v the average speed on the desired road. We estimated the constants as $F_N = 15\,000\text{kg}\,\text{m/s}^2$, $c_r = 0.015$, $A = 2.67\text{m}^2$, $c_w = 0.3$ and $\rho = 1.2\text{kg/m}^3$. The resulting cost function is defined as $\widetilde{C}(v,\ell) = (\ell \cdot c \cdot F(v))/\eta$ with ℓ denoting the length of the road, c denoting the cost of energy and η denoting the efficiency of the engine. We estimated $c = 0.041€/MJ$, this corresponds to a fuel price of about $1.42€/\ell$, and $\eta = 0.25$. For inner city roads, we multiply the cost by a factor of 1.5 to compensate for traffic lights and right of way situations. Note that this method is a good approximation for higher speeds but disregards the bad efficiency of the engine and transmission on low speeds. To get a more realistic model, we favor an average speed of 50 km/h and define our final weight function as

$$C(v,\ell) = \left\{ \begin{array}{ll} \widetilde{C}(50+\sqrt{50-v},\ell) & \text{if } v < 50 \\ \widetilde{C}(v,\ell) & \text{otherwise} \end{array} \right\} .$$

Figure 5.3 plots $C(v,1)$ against the travel speed.

Parameter Interval. A good parameter interval for the travel time and the energy cost function is between 0 and 0.1. For $p = 0$, we find the fastest path and for $p = 0.1$,

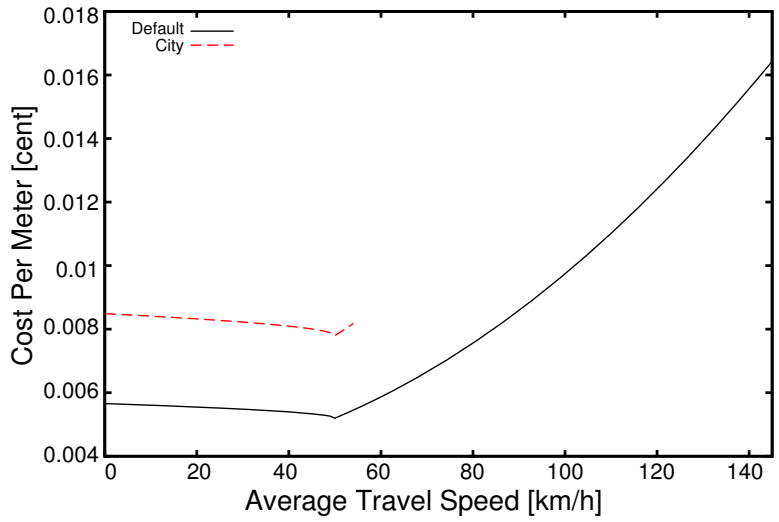


Figure 5.3: Energy costs per meter against the travel speed. The upper curve is for inner city roads.

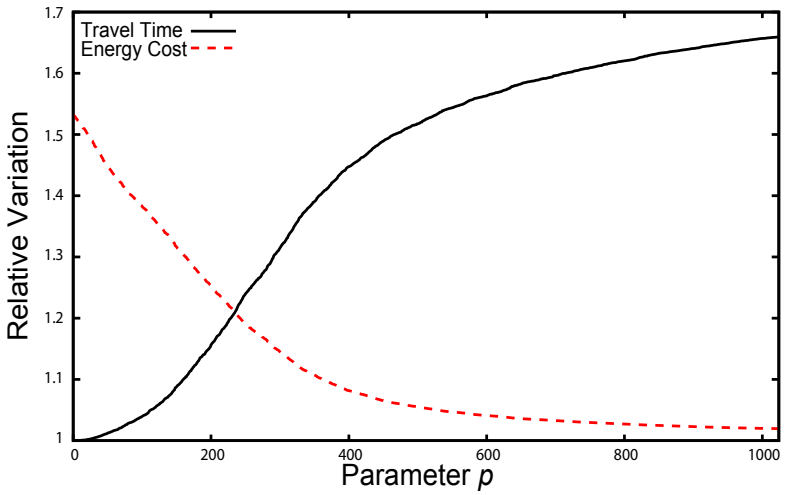


Figure 5.4: Average increase of the travel time and decrease of the energy cost as ratio to the best possible routes.

we observe an increase of the average travel time by 65% on GER. This increase is reasonable, and also Pareto-SHARC considers at most 50% increase.

Since we only support integer parameters, we scale all weights by a factor of 10 000 to be able to adjust the parameter with four digits after the decimal point. This results in a parameter interval of $[0, 1\,000]$, but actually we rounded to $[0, 1\,023]$. To prove the robustness of our approach, we additionally perform some experiments for the larger parameter interval $[0, 2\,047]$ using the same scale factor of 10 000.

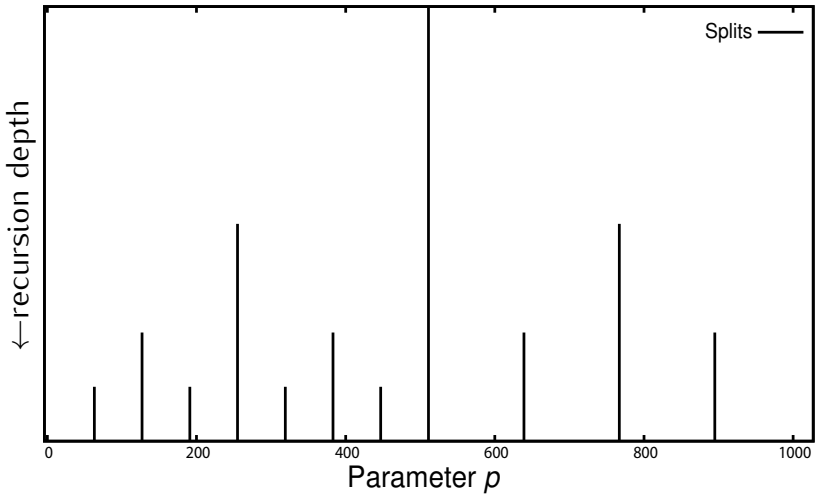


Figure 5.5: The vertical lines present the split positions. The higher a line, the lower is the recursion depth of the split.

Parameter Splitting. During the contraction, we maintain a threshold T; when there are more than T partial shortcuts since the last split, we trigger a split. After a split, we increase T since there are more shortcuts necessary in the upper hierarchy. We start with $T = 1.3\%$ of the number of edges in the input graph. After a split, we multiply T with 1.2. We do not split parameter intervals of size 16 or below.

We use only half split, preliminary experiments with variable split did not yield significant improvements. In Figure 5.5, the splits of the parameter interval are visualized. The longer a line, the earlier the split happened. We see that in the interval $[0,511]$ 7 splits happened whereas in $[512,1023]$ only 3 happen. This correlates quite well with the observations from Figure 5.4, where great changes are especially visible for the first half of the interval.

Performance. We summarize the performance of different variants of our algorithm in Table 5.6. When we only use contraction, preprocessing on GER takes 2.4 hours, resulting in an average query time of 2.9 ms. Note that the average is over 100 000 queries, where source, target and the value of p are selected uniformly at random. We split the precomputation time into the compute part and the I/O part. The I/O part is the time to write the intermediate results to disk when a split happens. You see that it can take a significant percentage of the overall precomputation time, up to one third, but can easily be avoided by using more main memory.

We usually select 64 *avoid* landmarks [72] per core. Compared to full contraction, a 5k uncontracted core has 12% better query time and significantly decreases the precomputation by one third. As expected, a 3k core results in even better query times, at the cost of precomputation time. However, switching to 32 landmarks is not significantly better for precomputation time and space, but increases the query time by 7%. Our best

Table 5.6: Preprocessing and query performance for different *graph*s. *param* specifies the number of different parameters x in the interval $[0, x-1]$. The *core* size and number of *landmark*s are given. The *preproc*essing time in hh:mm is split into the *compute* time and the *I/O* time needed for reading/writing to disk. Also the number of *split*s and the *space* consumption in Byte/node are given. The query performance is given as *query* time in milliseconds, *speed-up* compared to plain Dijkstra, number of *settled nodes* and number of *relaxed edges*.

		core/	preproc [hh:mm]		#	space	query	speed	settled	relaxed
graph	param	landmark	contr	IO	splits	[B/node]	[ms]	-up	nodes	edges
GER	1 024	-/-	-	-	-	60	2.04 s	1	2.36 M	5.44 M
GER	1 024	0/0	1:54	0:29	11	159	2.90	698	579	4 819
GER	1 024	10k,C/64	2:06	0:29	11	183	0.63	3 234	170	2 059
GER	1 024	3k/64	1:13	0:29	11	164	2.33	874	620	9 039
GER	1 024	5k/32	1:05	0:30	11	161	2.76	738	796	11 137
GER	1 024	5k/64	1:06	0:30	11	167	2.58	789	735	10 191
GER	2 048	5k/64	1:30	0:37	14	191	2.64	771	734	9 835
EUR	1 024	-/-	-	-	-	59	15.1 s	1	6.08 M	13.9 M
EUR	1 024	5k/64	12:55	2:32	11	142	6.80	2 226	1 578	32 573
EUR	1 024	10k/64	11:58	2:31	11	144	8.48	1 784	2 151	39 030
EUR	1 024	10k,C/64	18:37	2:35	11	145	1.87	8 097	455	7 638
WEU	16	10k,C/64	1:00	0:10	0	60	0.42	14 427	270	2 103
WEU	1 024	10k,C/64	5:12	1:12	7	151	0.98	6 183	364	3 360

query times are with a 10k contracted core yielding speed-up of more than 3 000.

You cannot directly compare our performance to previously published results of single-criteria CH, since the performance heavily depends on the edge weight function. We computed single-criteria CH for GER with $p = 0$, $p = 1000$ and for $p = 300$, one of the most difficult parameters from Figure 5.8. The preprocessing time varied by about 100% between these parameters and the query time even by 270%. Only space consumption[1] is quite constant, it changed by less than 3% and is around 22 B/node. We compare our 5k/64 core to economical CH [59] as both have the best preprocessing times. Our space consumption is a factor 7.6 larger, however we could greatly reduce space in relation to our 1 024 different parameters, even below the number of 12 different cores that exist due to 11 splits. Also, we could compute 18.9 different hierarchies within the time needed by our new algorithm. For this comparison, we ignored the preprocessing I/O time since the single-criteria CH also needs to be written to disk. The reduction in preprocessing time is therefore not as large as for space, but still good. However, our efficient preprocessing comes at the cost of higher query times. Single-criterion CH has 0.82 ms query time on average but our query time is about three times larger. One reason for our higher query time is the shared node order within a parameter interval that is

[1] do not mistake it for space overhead

not split, but we also have a larger constant factor because of the data structure: there is an additional level of indirection due to the buckets, and we store two weights and a parameter interval per edge, resulting in more cache faults. The frequently occurring weight comparisons are also more expensive and noticeable, since multiplications and additions are necessary. But e. g. for web services, the query time is still much lower than other delays, e. g., for communication latency. Using landmarks on a contracted core would yield even faster query times than single-criteria CH, but this would not be a fair comparison as we should use landmarks there as well.

Our algorithm scales well with the size of the interval. Increasing it to twice the size only increases the preprocessing time by 32% and the space by 14% without affecting query time much. We also did some experiments on EUR that worked well for GER, but we could not perform a thorough investigation, since the precomputation took very long. The query times are very good, yielding speed-ups of more than 8 000. Better speed-ups are expected for larger graphs, as for single-criteria CH, too. The space consumption is even better, the dense road network of GER is more difficult than the rest of EUR. Preprocessing time is however super-linear, but we expect that tuning the node ordering parameters and the split heuristic will alleviate the problem.

Edge Buckets. As already explained earlier, the number of scanned edges has a large impact on the quality of the query. When the number of memory accesses for non-necessary edges is large enough, it pays of to omit the respective edges, even if an additional level of indirection has to be used for some of the edges. By default, for each parameter interval that was not further split, we have one bucket, i. e. with 11 splits we have 12 buckets that split the parameter interval as in Figure 5.5. To investigate the effect further, we take two computed hierarchies from Table 5.6, remove the original buckets and use different numbers of buckets that split the parameter interval in equally spaced pieces. The results are in Table 5.7 and show the different trade-offs between query time and space consumption. If we compare the data to Table 5.6, we can see that even with buckets, the number of scanned edges is more than two times larger than the number of relaxed edges. Comparing the different numbers of buckets, we notice that around a quarter of all edges would be stored in all buckets. Therefore, storing them separately helps to control the space, since buckets already consume a major part. About 40% of our space is due to buckets, but we get almost the same percentage as improvement of the query time. Using just two buckets even increases the query time as another level of indirection is necessary to address the edges in the buckets even though we only halve the number of scanned edges. Our choice of 12 buckets is in favor of fast query times, more buckets bring only marginal improvements. Figure 5.8 visualizes the effects of buckets for different values of p. We see that ≥ 4 buckets improve the query time for all parameters but in the interval $[0,511]$ more buckets are necessary than in $[512,1023]$ as 12 buckets show the most uniform performance over the whole interval. When we ignore the different bucket sizes, we also note that our algorithm achieves the best query times for $p = 0$, when we optimize solely for travel time. Therefore, our performance

depends on the chosen value of p and furthermore on the chosen edge weights functions as they have a strong impact on the hierarchical structure of the network.

Table 5.7: Query performance for different numbers of edge buckets.

graph	core/ landmark	buckets	edges in all buckets	query [ms]	scanned edges	memory overhead [B/node]
GER	10k,C/64	1	100.0%	0.96	41 613	0
GER	10k,C/64	2	26.9%	1.02	23 486	19
GER	10k,C/64	4	26.5%	0.81	13 388	28
GER	10k,C/64	8	26.4%	0.69	7 757	49
GER	10k,C/64	12	26.4%	0.63	5 769	71
GER	10k,C/64	16	26.4%	0.62	5 270	96
GER	3k/64	1	100.0%	3.75	170 483	0
GER	3k/64	2	27.5%	3.90	94 876	18
GER	3k/64	4	27.0%	3.18	53 082	28
GER	3k/64	8	26.9%	2.60	29 866	48
GER	3k/64	12	26,9%	2.33	21 694	70
GER	3k/64	16	26.9%	2.32	20 754	95

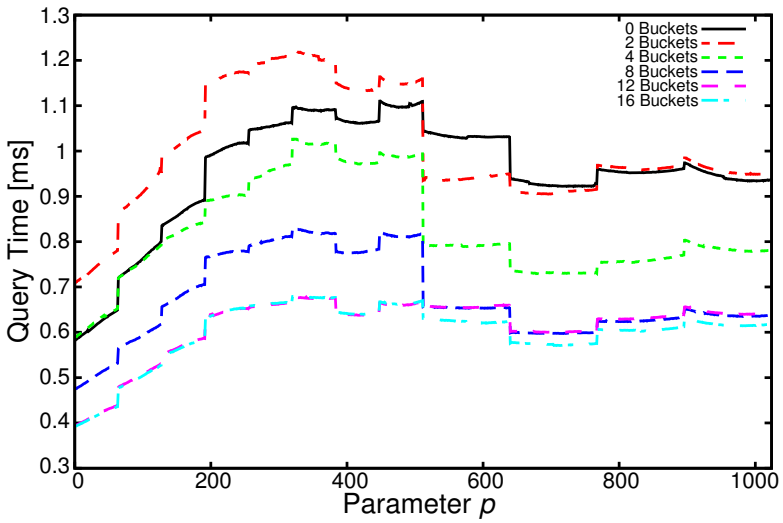


Figure 5.8: Average query time in dependence of the value of p and the used number of buckets. We used the GER graph with 10k contracted core and 64 landmarks.

Profile Query. Utilizing our flexible fast query algorithm, we can also compute the shortest paths for *all* values of p. We get about $k = 31$ different paths on average and

need 88 queries for different values of p to compute them (Table 5.9). This is close to the bound of $3k-2=91$ queries, which are theoretically necessary. So applications that need all shortest paths between two locations should use our fastest version.

Table 5.9: Performance of the profile search in different versions.

graph	core/ landmark	found paths	flexible queries	time [ms]
GER	10k,C/64	30.7	88.1	59.4
GER	3k/64	30.7	88.1	307.0
GER	5k/64	30.7	88.1	349.6

When a reasonable subset of all such paths is sufficient, our sampling approach provides better performance. A possible application could be to present a number of samples for the user to choose from and maybe calculate more detailed information on demand for certain parameter intervals of the presented choices. In Table 5.10, we give an overview over the performance of our sampling version of the profile query, with the main focus on the number of found routes and running time for different numbers of equidistant sample values of p. As expected, we see a linear increase of the query time compared to the number of performed queries. Up to 9 samples, we get a new path with almost every query. However, the more samples we choose, the more queries are done in vain. Still, we can choose at query time very fine-grained how many different paths we want. This is a clear advantage over Pareto-SHARC, were the average number of target labels (= #paths) is limited to 5–6 since otherwise the precomputation and query time gets out of hand.

Table 5.10: Performance of sampled profile search on GER with a contracted core of 10 000 nodes using 64 avoid landmarks.

samples	flexible queries	found paths	time [ms]
2	1.8	1.8	1.0
3	2.7	2.7	1.7
5	4.4	4.3	2.9
9	8.0	7.2	5.5
17	15.8	11.4	10.2
33	31.4	17.3	18.9
65	58.3	22.8	34.3
129	95.9	27.1	57.1

Approximated Profile Query. Another possibility to reduce the profile query time is using ε-approximation. We prune our profile query when all paths that we may miss

are within an ε factor of the currently found paths. By that, we balance the query time without missing significant differently valued paths as it may happen with sampling. In Table 5.11 we see the same ratio between found paths and query time as for the sampling approach. Many of the 31 different paths are very close as for a small $\varepsilon = 1\%$ the number of paths is cut in halves. Still, there are significant differences in the paths as even $\varepsilon = 16\%$ still has more than 4 different paths.

Table 5.11: Performance of the approximated profile query on GER with a contracted core of 10000 nodes an 64 avoid landmarks.

ε	found paths	flexible queries	time [ms]
0	30.7	88.3	59.4
0.00125	26.5	55.0	36.4
0.0025	23.7	46.0	30.0
0.005	20.4	36.9	24.4
0.01	17.0	28.3	18.9
0.02	13.3	20.1	13.5
0.04	9.7	12.9	8.7
0.08	6.5	7.6	5.3
0.16	4.2	4.4	3.0
0.32	2.7	2.7	1.7

Comparison with Previous Work. Even though Pareto-SHARC [45] can handle continent-sized networks only heuristically and we have exact shortest paths, we perform very well in comparison since we do not rely on Pareto-optimality. We used the same network of Western Europe (WEU) and costs as Pareto-SHARC. With 16 values of p, the average travel time increases by 4.3%. This case allows in our opinion the closest comparison to Pareto-SHARC with simple label reduction with $\varepsilon = 0.02$ (4:10 hours preprocessing and 48.1 ms query). Our precomputation is 3.6 times faster (see Table 5.6) and our profile query (2 ms) is more than 24 times faster. For 1024 different values of p, the average travel time increases by 43.4%, that might be close to heuristic Pareto-SHARC with strong label reduction ($\varepsilon = 0.5$ and $\gamma = 1.0$, 7:12 hours preprocessing and 35.4 ms query). On WEU, we could not reach an average increase of 50%, even doubling the values of p yields less than 44% travel time increase. We need 12% less preprocessing time, and an approximate profile query with $\varepsilon = 0.01$ returning 5.7 different paths (6.2 ms) is 5.7 times faster. Contrary to Pareto-SHARC, we can also just compute a single path for a single value of p, taking just 0.4 ms. Furthermore, we provide more flexibility with 12.5 different paths available on average over the whole parameter interval. But compared to GER, we have less different paths. This is due to the different "costs": Pareto-SHARC uses a simple model to calculate fuel- and toll-costs whereas our model of the energy cost is based upon laws of physics. The only downside of our

algorithm is that we need more space than Pareto-SHARC (22.5 B/node preprocessing + 23.7 B/node input graph for Pareto-SHARC). However, we can also provide more different routes. In conclusion, our linear combination of two edge weight functions allows a faster computation than a combination based on Pareto-optimality. We scale better since we can split parameter intervals, whereas Pareto-optimal weights naturally cannot be split and distributed to buckets. Adapting the SHARC algorithm to our flexible scenario is possible. However, such a Flexible-SHARC algorithm also needs, for efficiency, necessity intervals, edge buckets, or maybe multiple arc flags for different parts of the parameter interval. Therefore it is not clear, whether it will turn out to be more space-efficient than our CH-based algorithm.

5.3 Edge Restrictions

Edge restrictions extend a static graph by *threshold* functions on edges, for example bridge height. A query then specifies *constraints*, for example vehicle height, and omits all restricted edges for the shortest-path computation. More formally, edge restrictions extend a static graph $G = (V,E)$ by an r-dimensional *threshold* function vector $a = \langle a_1, \ldots, a_r \rangle$ such that $a_i : E \to \mathbb{R}_+ \cup \{\infty\}$ assigns thresholds to edges (Section 2.2.3). The query parameter is a vector $p = \langle p_1, \ldots, p_r \rangle$ that *constraints* edges according to their threshold following Definition 5.12.

Definition 5.12 *A* flexible query *in the scenario with edge restrictions computes the shortest-path distance $\mu_p(s,t)$ between a source node s and a target node t for* query constraint parameters $p = \langle p_1, \ldots, p_r \rangle$ *such that for every edge e of the according shortest path holds $p_i \leq a_i(e)$, $i = 1..r$.*

5.3.1 Preliminaries

In this section, we define some mathematical notations for working with thresholds on edges and paths.

Definition 5.13 *Given two edges e and e', we define the* minimum threshold *vector for these two edges to be:*

$$a(e) \curlywedge a(e') = \langle min(a_1(e), a_1(e')), min(a_2(e), a_2(e')), \ldots, min(a_r(e), a_r(e')) \rangle$$

Definition 5.14 *We define the* maximum threshold *vector analogously:*

$$a(e) \curlyvee a(e') = \langle max(a_1(e), a_1(e')), max(a_2(e), a_2(e')), \ldots, max(a_r(e), a_r(e')) \rangle$$

The minimum/maximum threshold operators define the most/least restrictive set of threshold values between the two edges' threshold function vectors, respectively.

Definition 5.15 *For any path $P = \langle e_1, e_2, \ldots, e_k \rangle$, we define the* path threshold *as the minimum threshold over all of its edges (e. g., see Figure 5.12):*

$$a(P) = a(e_1) \curlywedge a(e_2) \curlywedge \ldots \curlywedge a(e_k)$$

The path threshold operation effectively defines the most restrictive set of possible query constraint parameters for which the associated path will still be unrestricted.

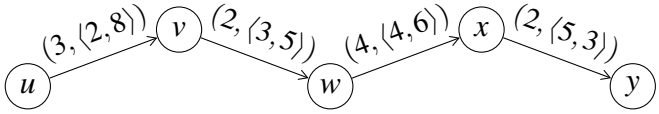


Figure 5.12: For the path $P = \langle (u,v), (v,w), (w,x), (x,y) \rangle$, we have $a(P) = \langle 2, 3 \rangle$.

Definition 5.16 *Given two threshold vectors a and a', we define a* weak dominance *relation as follows:*

$$a \preceq a' \Leftrightarrow \forall i \in \{1,\dots,r\}, a_i \leq a'_i$$
$$a \not\preceq a' \Leftrightarrow \exists i \in \{1,\dots,r\}, a_i > a'_i$$

In this context, we say that a' weakly dominates a ($a \preceq a'$) if it is no more restrictive than a for all possible restrictions. Likewise, we say that a is non-dominated by a' ($a \not\preceq a'$) if it is less restrictive than a' for at least one restriction. We additionally apply this notation to reflect the relation between any query constraint parameters $p = \langle p_1,\dots,p_r\rangle$, and the threshold function vectors. For example, we say that $p \preceq a(e) \Leftrightarrow \forall i \in \{1,\dots,r\}, p_i \leq a_i(e)$ to indicate that p is not restricted by $a(e)$. By that we can more easily write the static graph $G_p = (V, E_p)$ introduced in Section 2.2.3 by $E_p := \{e \in E \mid p \preceq a(e)\}$. A similar notation holds for $\not\preceq$ as well.

5.3.2 Node Contraction

In the following, we show how to augment the concept of node contraction to our flexible scenario with edge restrictions. When contracting a node v, we need to consider edge restrictions. For each pair of edges (u,v) and (v,w) in the remaining (uncontracted) graph the potential shortcut (u,w) represents the path $\langle u,v,w\rangle$ with path threshold $p = a(u,v) \curlywedge a(v,w)$. To avoid the shortcut, we need to find a witness path whose *witness path threshold* weakly dominates p. We do this by performing the witness search in the graph G_p that only contains edges that weakly dominate p, see Figure 5.13. As before with the static node contraction, we then only add a shortcut edge (v,w) if the resulting path P has greater length than the path $\langle u,v,w\rangle$, and we omit it otherwise.

Note that contrary to the static scenario, we cannot omit *parallel edges*, see Example 5.17. We need to keep parallel edges for each Pareto-optimal pair of threshold vector

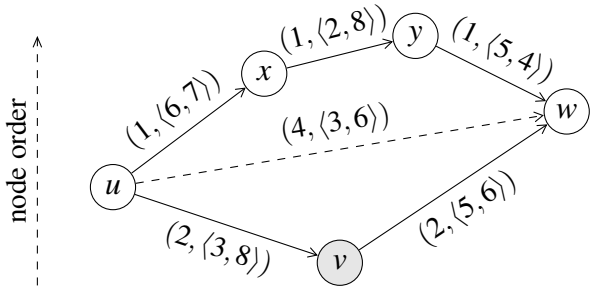


Figure 5.13: Contracting node v for flexible edge restrictions. The shortcut edge is represented by a dashed line. In this case, we have that $p = a(u,v) \curlywedge a(v,w) = \langle 3,6\rangle$. Since there is no valid path P for this witness search which ignores node v (i. e., edge (x,y) is restricted for $p_1 = 3$ and edge (y,w) is restricted for $p_2 = 6$), a shortcut edge (u,w) must be added with weight $c(u,v) + c(v,w) = 4$ and threshold function vector p.

and edge weight. However, we keep identifying edges by their two endpoints in cases where the particular edge is clear from the context.

Example 5.17 *Assume we have an edge with weight 2 and threshold $\langle 2,6\rangle$, and a parallel edge with weight 4 and threshold $\langle 3,8\rangle$. We cannot just keep the edge with weight 2, as a query with constraints $\langle 3,8\rangle$ must not use this edge. Therefore, parallel edges are required.*

The formal definition for the correctness of our node contraction states Lemma 5.18. The proof of this lemma follows directly of our definition of node contraction and the transitivity of the threshold dominance.

Lemma 5.18 *Consider the contraction of node v in the scenario with flexible edge restrictions. Let u, w be two uncontracted neighbors of v with edge (u,v), (v,w). Let p be a set of query constraint parameters with $p \preceq a(u,v) \curlywedge a(v,w)$. If there exists no witness path P with $c(P) \leq c(\langle u,v,w\rangle)$ and $p \preceq a(P)$, a shortcut of weight $c(\langle u,v,w\rangle)$ and threshold vector $a(u,v) \curlywedge a(v,w)$ is added.*

Multi-Target Witness Search. To establish a minimal number of possible shortcut edges for a given node ordering, in general, a witness search must be carried out separately for every pair of neighbors for which a shortcut may need to be added. This is necessary, since the exact constraints applied to any particular witness search will depend upon the threshold values of the specific pair of edges being bypassed by a given (potential) shortcut edge. Therefore, when contracting a node v, where $S = \{(u,v) \in E \mid u \text{ contracted after } v\}$ and $T = \{(v,w) \in E \mid v \text{ contracted before } w\}$, the naïve algorithm must perform a total of $|S| \cdot |T|$ separate witness searches[2]. The overall efficiency of the contraction of v can be improved by instead performing only a single witness search from the source u of each incoming edge $(u,v) \in S$ until all targets w of outgoing edges $(v,w) \in T, w \neq u$ have been settled, or until a distance of $c(u,v) + max\{c(v,w) \mid (v,w) \in T, w \neq u\}$ has been reached. We use the maximum threshold of all the outgoing edges in T to obtain the path threshold $p = a(u,v) \curlywedge \left(\curlyvee_{(v,w)\in T, w\neq u} a(v,w)\right)$ (e. g., see Figure 5.14). We perform the multi-target witness search in G_p. However, this does not affect the correctness of Lemma 5.18, since the witness search constraint parameters p are guaranteed to be as or more restrictive than the constraint parameters for any explicit pair with (u,v) as the incoming edge. Therefore, any resulting witness paths are still valid, even though this approach may result in the addition of unnecessary shortcuts. As we will see in later experiments, this multi-target approach scales much better in practice.

We limit the witness search to 7 hops, cf. Section 3.2.1. This may add superfluous shortcuts but does not affect the correctness.

[2]Pairs $(u,v) \in S, (v,w) \in T$ where $u = w$ may be ignored.

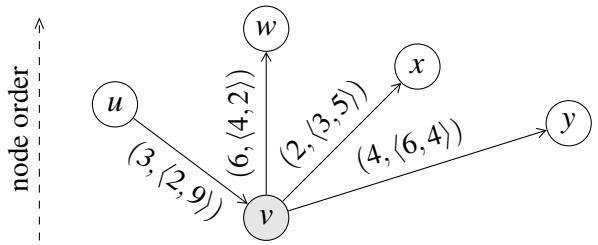


Figure 5.14: Multi-target witness search from u to $\{w,x,y\}$. In this scenario, the query constraint parameters are defined as $p = a(u,v) \curlywedge (a(v,w) \curlyvee a(v,x) \curlyvee a(v,y)) = \langle 2,9\rangle \curlywedge (\langle 4,2\rangle \curlyvee \langle 3,5\rangle \curlyvee \langle 6,4\rangle) = \langle 2,9\rangle \curlywedge \langle 6,5\rangle = \langle 2,5\rangle$.

Edge Reduction. The edge reduction (Section 3.2.1) removes edges that do not represent shortest paths. We adapt this idea to our flexible scenario as follows. After performing a multi-target witness search (discussed above) from the source u of some incoming edge (u,v), we have computed a shortest-path tree rooted at node u. Let P_x be the path in the tree from root node u to node x. Then, for all outgoing edges (u,x) from u, we may remove edge (u,x) if $c(P_x) \leq c(u,x)$, $P_x \neq \langle (u,x)\rangle$ and $a(u,x) \preceq a(P_x)$. In this case, it is easy to show that edge (u,x) can be replaced by P_x in any shortest path. Therefore, edge (u,x) can easily be removed in this scenario without affecting the correctness of any subsequent shortest-path queries.

Witness Restrictions. Based on the preprocessing described so far, a query algorithm tends to have a much larger search space (i. e., a larger number of relaxed edges and settled nodes) for less restricted queries than for more restricted queries. This comes from the fact that more restricted queries will filter out more edges from the search. However, a lot of shortcuts are added during the node contraction that are only necessary for more restricted queries, and for less restricted queries there would be a witness preventing the shortcut. In the flexible scenario with two edge weights, we computed a *necessity interval* for each edge so that only necessary edges are relaxed (Section 5.2.1). For the current scenario, we developed a corresponding concept, however tailored to edge restrictions. We call this new concept *witness restrictions*. It stores the information about these witnesses for less restricted queries with the shortcuts. More precisely, this information is a set of witness path thresholds. There can be more than one witness path threshold for a given shortcut edge, see Figure 5.15. This is contrary to the scenario with two edge weights, where a single necessity interval stores all corresponding information.

Note that the witness paths considered for the set of witness path thresholds must be strictly shorter than the shortcut. This is necessary, as the shortcut may be used as a witness when all the interior nodes on the witness path get contracted before its source and target node. Also note that it is sufficient to store a Pareto-optimal set of witness path thresholds.

However, storing a whole set of witness path thresholds would require a variable

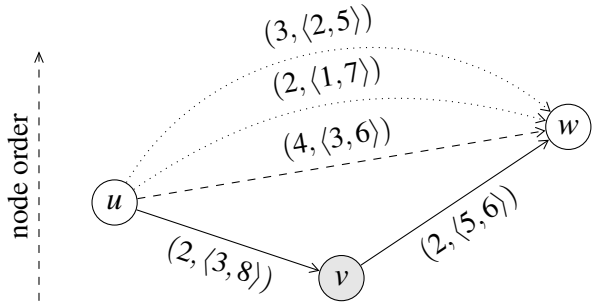


Figure 5.15: Contracting node v with two witnesses (dotted) available for a subset of potential query constraint parameters. The shortcut edge is represented by a dashed line. The witness path thresholds of the shortcut (u,w) are $\{\langle 1,7\rangle, \langle 2,5\rangle\}$. The shortcut is necessary, e. g. for a query with constraint parameters $\langle 3,5\rangle$, as this restricts both potential witnesses. But, for example, a query with constraint parameters $\langle 0,6\rangle$ or $\langle 2,5\rangle$ does not need the shortcut, as a witness is available. Note that we cannot just store the maximum of the two witness paths thresholds $\langle 2,7\rangle$, as for this set of query constraint parameters, no witness is available.

amount of storage overhead per edge, which is impractical. Also, it is too time-consuming to compute all Pareto-optimal witness path thresholds. Therefore, for practicality, we will only store a single witness path threshold $a^*(e)$ with a shortcut e. For non-shortcuts e, we set $a^*(e) = \langle -\infty, -\infty, \ldots, -\infty\rangle$. During preprocessing, when a new shortcut edge $e = (u,w)$ is added, we then perform an additional unconstrained shortest-path query that tries to find the shortest overall witness path $P_{u,w}$. We then set $a^*(u,w) = a(P_{u,w})$ and continue as before. Note that in our actual implementation, we delay the computation of $a^*(e)$ until one of its endpoints gets contracted. This improves the preprocessing performance, as the computation is then executed on a smaller remaining graph, and the edge reduction technique may removed e in between.

During the execution of a query with query constraint parameters p, we can prune the relaxation of an edge e with $p \preceq a^*(e)$. The correctness of the query algorithm is not affected due to Lemma 5.19.

Lemma 5.19 *For an edge $e = (u,w) \in E$ and a set of query constraint parameters p with $p \preceq a^*(e)$ holds $\mu_p(u,w) < c(e)$.*

Proof. Follows from the fact that witness paths considered to compute $a^*(e)$ must be strictly shorter than the edge e. □

As will be seen in later experiments, using witness restrictions can result in significantly smaller search space sizes and query times, especially for less restricted queries. The only downside of witness restrictions is that they increase the space required to store an edge. We therefore also propose to store them only for edges inside a core of the most important nodes, similar to the core-based landmarks described in Section 3.4.

Node Ordering. As described in Section 3.2, we assign each node v a *priority* on how attractive it is to contract v. Then, we contract a most attractive node and update the priorities of the remaining nodes. We repeat this until all nodes are contracted. This results in an ordering of the nodes and, at the same time, adds all necessary shortcuts to the graph. The priority is a linear combination of two terms. The first term is the *edge difference* between the number of necessary shortcuts to contract v and the number of incident edges to remaining nodes. The second term is the sum of the *original edges* represented by the necessary shortcuts to contract v. We weight the first term with 1 and the second term with 2 (these weights were determined from initial experiments on graphs with binary restrictions [122]). We use the heuristics for priority updates, i. e. updating only neighbors of a contracted node and using *lazy updates*.

5.3.3 A* Search using Landmarks (ALT)

We can easily combine our algorithm based on node contraction with ALT [69]. We described the basic concept in Section 3.4. The ALT technique has been previously studied within the context of dynamic graphs [44], where it is noted that the potential functions computed from landmarks in ALT remain correct for dynamic scenarios in which edge weights are only allowed to increase from their original value. Therefore, it is easy to see that ALT remains correct even in the current flexible scenario, since restricting an edge is equivalent to increasing its weight to infinity for the duration of the query.

Multiple Landmark Sets. Using ALT landmark distances can result in ineffective potential functions for very heavily-constrained shortest-path queries [122]. This is due primarily to the fact that, even though the potential functions remain correct, the resulting lower bounds become much weaker (i. e., less accurate) when large numbers of edges become restricted.

Therefore, in an attempt to alleviate this problem, we propose the new concept of using *multiple landmark sets* $\mathscr{L} = \{L_1, L_2, \ldots, L_k\}$. In this context, we broadly define a landmark set $L \subseteq V$ as being a set of landmarks created specifically for a single set of query constraint parameters $p_L = \langle p_1, p_2, \ldots, p_r \rangle$. The distances for the landmark set L are based on the edge-restricted graph determined by the query constraints of p_L.

One possibility to support this for flexible edge restrictions would be to establish a unique landmark set for all possible combinations of query constraint values. However, this is clearly infeasible for any real-world implementation. A more realistic approach would therefore be to establish a unique landmark set specific to each unique threshold function a_i in the threshold function vector for the graph. That is, $\mathscr{L} = \{L_1, L_2, \ldots, L_r, L_{r+1}\}$, such that $\forall i \in \{1, \ldots, r\}$, $p_{L_i} = \langle p_1 = 0, \ldots, p_{i-1} = 0, p_i = \infty, p_{i+1} = 0, \ldots, p_r = 0 \rangle$ (i. e., each landmark set L_i restricts only edges with finite threshold values for a_i). Landmark set L_{r+1} is established for the fully *unrestricted* set

of query constraint parameters $p_{L_{r+1}} = \langle 0, 0, \ldots, 0 \rangle$ to ensure that there is always at least one set of valid landmarks to choose from.

5.3.4 Query

After preprocessing is complete, that is, all necessary shortcuts are added to G, we can use an augmented version of the query algorithm described in Sections 3.2.2. It runs on the search graph G^*, defined just as in Sections 3.2.2. We give pseudo-code in Algorithm 5.6. The only necessary augmentation is to restrict edges using the query constraint parameters p (Line 14).

Algorithm 5.6: FlexibleEdgeRestrictedCHQuery(s, t, p)

input : source s, target t, query constraint parameters p
output : shortest-path distance δ

1 $\overrightarrow{\delta} := \langle \infty, \ldots, \infty \rangle$; // tentative forward distances
2 $\overleftarrow{\delta} := \langle \infty, \ldots, \infty \rangle$; // tentative backward distances
3 $\overrightarrow{\delta}(s) := 0$; // forward search starts at node s
4 $\overleftarrow{\delta}(t) := 0$; // backward search starts at node t
5 $\delta := \infty$; // tentative shortest-path distance
6 $\overrightarrow{Q}$.update(0, s); // forward priority queue
7 $\overleftarrow{Q}$.update(0, t); // backward priority queue
8 $\sim := \rightarrow$; // current direction
9 **while** $(\overrightarrow{Q} \neq \emptyset)$ **or** $(\overleftarrow{Q} \neq \emptyset)$ **do**
10 **if** $\delta < \min\left\{\overrightarrow{Q}.\min(), \overleftarrow{Q}.\min()\right\}$ **then break**;
11 **if** $\overrightarrow{\widetilde{Q}} \neq \emptyset$ **then** $\sim := \neg \sim$; // change direction, $\neg \leftarrow = \rightarrow$ and $\neg \rightarrow = \leftarrow$
12 $(\cdot, u) := \widetilde{Q}$.deleteMin(); // u is settled
13 $\delta := \min\left\{\delta, \overrightarrow{\delta}(u) + \overleftarrow{\delta}(u)\right\}$; // u is potential candidate
14 **foreach** $e = (u, v) \in E^*$ **with** $\sim(e)$ **and** $p \preceq a(e)$ **and** $p \not\preceq a^*(e)$ **do** // relax edges
15 **if** $\left(\widetilde{\delta}(u) + c(e)\right) < \widetilde{\delta}(v)$ **then** // shorter path via u?
16 $\widetilde{\delta}(v) := \widetilde{\delta}(u) + c(e)$; // update tentative distance
17 $\widetilde{Q}$.update$\left(\widetilde{\delta}(v), v\right)$; // update priority queue
18 **return** δ;

Theorem 5.20 *Given a source node s, a target node t and query constraint parameters p, our CH query for the flexible scenario with edge restrictions computes* $\mu_p(s,t)$.

Proof. Let us construct a static graph induced by the edges $\{e \in E \mid p \preceq a(e)\}$. Obviously, our query for parameter value p corresponds to the static CH query on this constructed graph. Lemma 5.18 proves that we add all necessary shortcuts for p on paths that could

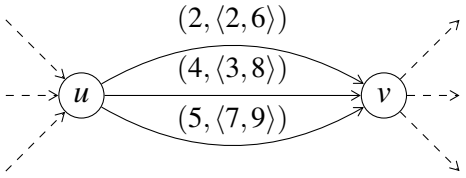


Figure 5.16: Skipping parallel edges. Edges to a given target node are accessed in order of increasing weight to allow pruning. For example, for a query with constraint parameters $p = \langle 1, 3 \rangle$, we may skip the bottom two parallel edges from node u to node v, since we can successfully relax the topmost edge.

be part of a shortest path w. r. t. the query constraint parameters p. We can further ignore edges e with $p \preceq a^*(e)$ as they would never be part of a shortest path with query constraint parameters p (Lemma 5.19). Therefore, we can deduce from Corollary 3.2 that our algorithm computes $\mu_p(s,t)$. □

Parallel Edge Skipping. Due to the presence of multi-edges within the resulting CH graphs, a naïve shortest-path query on the upward/downward edge-restricted graph may perform redundant or unnecessary work on any parallel edges during a given search. However, if we sort and store the adjacent edges of a given node first by their target node and then by their weight, then, when we relax the first valid (i. e., unrestricted) edge leading to a given target node, we can automatically skip over any remaining parallel edges leading to that same target node, since no remaining parallel edges to that node can improve the current path distance (see Figure 5.16). This will save on unnecessary (and ultimately, unsuccessful) attempts to update tentative shortest-path distances. Additionally, and perhaps more importantly, when incorporating goal-directed search into the query, skipping of parallel edges will also allow us to save on redundant calls to compute the potential function value for the same target node, which can be relatively expensive.

Combination with ALT. We perform the two-phased query of Section 3.4, but consider edge restrictions. Instead of the static CH query algorithm we use the algorithm augmented to our flexible scenario.

Using this concept of multiple landmarks, for a given query with constraint parameter p, we may then choose only from the set of landmarks $\mathscr{L}_p = \{L \in \mathscr{L} \mid p_L \preceq p\}$. That is, we may select only a set of landmarks L, whose associated query constraint parameters p_L are no more restrictive than the incoming query constraint parameters p. By that, we maintain valid lower bounds for the resulting potential function values. Using multiple landmark sets allows us to derive better potential functions, in general, by more closely approximating the shortest path distances associated with the dynamic query constraints of p.

Instead of deriving the lower bounds from all landmarks in $\mathscr{L}_p$, we only use a subset of 4 landmarks that give the highest lower bounds on the s-t distance [69]. This speeds

up the overall query, as the computation of the lower bounds is much faster but still provides good lower bounds.

In the ALT algorithm, the heuristic potential function serves to establish a lower bound on the possible shortest-path distance of a given query. Since we maintain a tentative upper bound on the shortest-path distance for our current query, we may also skip over any nodes whose resulting key value (which is a lower bound on the length of a path that visits that node) is greater than or equal to the current upper bound seen thus far [72]. This simple optimization can also be seen to have a significant impact on the resulting shortest-path query times.

5.3.5 Experiments

Environment. All experiments were carried out on a 64-bit server machine running Linux CentOS 5.3 with 2 quad-core CPUs clocked at 2.53 GHz with 72 GiB RAM (although only one core was used per experiment). All programs were written in C++ and compiled using the GNU C++ compiler 4.1.2 with optimization level 3.

Test Instances. Experiments were carried out on two of the largest available real-world road networks: a graph of North America[3] (NAV-NA) and a graph of Europe[4] (NAV-EU). NAV-NA has a total of 21 133 774 nodes and 52 523 592 directed edges, while NAV-EU has 40 980 553 nodes and 94 680 598 directed edges. Table 5.17 summarizes the different real-world restrictions supported by each graph dataset. Both datasets (including restrictions) were derived from NAVTEQ transportation data products, under their permission. Of the 18 unique restrictions, only two of these are true parameterized restrictions: Height and Weight Limit. The remaining 16 restrictions are binary restrictions only. For the parameterized restrictions, the data distinguishes between 29 different height limit values and 57 different weight limit values.

Unless otherwise stated, all query performance results are averaged over the shortest-path distance queries between 10 000 source-target pairs. Source and target node are selected uniformly at random. We performed for each source-target pair an unrestricted and a fully-restricted query and we report the mean performance. This covers both ends of the restriction spectrum.

[3]This includes only the US and Canada.

[4]This includes Albania, Andorra, Austria, Belarus, Belgium, Bosnia and Herzegovina, Bulgaria, Croatia, Czech Republic, Denmark, Estonia, Finland, France, Germany, Gibraltar, Greece, Hungary, Ireland, Italy, Latvia, Liechtenstein, Lithuania, Luxembourg, Macedonia, Moldova, Monaco, Montenegro, Netherlands, Norway, Poland, Portugal, Romania, Russia, San Marino, Serbia, Slovakia, Slovenia, Spain, Sweden, Switzerland, Turkey, Ukraine, United Kingdom, and Vatican City.

Table 5.17: Supported restriction types for both graphs.

	edges	
restrictions	NAV-NA	NAV-EU
Ferry	2 610	11 334
Toll Road	47 388	237 304
Unpaved Road	3 645 458	14 434 228
Private Road	1 662 314	1 957 380
Limited Access Road	682 396	N/A
4-Wheel-Drive-Only Road	139 284	N/A
Parking Lot Road	160 850	N/A
Hazmat Prohibited	45 950	N/A
All Vehicles Prohibited	64 414	2 326 232
Delivery Vehicles Prohibited	148 010	3 674 338
Trucks Prohibited	475 472	5 347 498
Taxis Prohibited	147 628	3 765 140
Buses Prohibited	151 272	3 811 704
Automobiles Prohibited	114 192	3 772 628
Pedestrians Prohibited	1 253 030	1 653 448
Through Traffic Prohibited	2 050 562	7 210 664
Height Limit	23 873	N/A
Weight Limit	24 627	N/A

Engineering an Efficient Algorithm. There is a significant performance difference between a basic and an optimized implementation[5]. We analyze this in Table 5.18 for NAV-NA. Single-target (i. e., pairwise-edge) witness search provides no feasible precomputation, as it does not finish within 3 days. With multi-target witness search, we significantly decrease precomputation time to 16 hours. Additionally using edge reduction decreases the precomputation to 7 hours, and also reduces the query time from 7.3 ms to 4.6 ms. Skipping parallel edges reduces the number of relaxed edges by 70%. But as we still need to traverse the edges in the graph, and of course the same number of nodes is settled, the query time is only reduced by 7%. Using witness restrictions increases the precomputation by less than 20 minutes and improves the query time by 22%. However, it increases the space-consumption by 9 B/node. Using witness restrictions only on a core, we even get a slightly improved query time and virtually no space overhead for witness restrictions. It seems that the time needed to evaluate the witness restrictions outside the core is higher than the time saved from pruning some shortcuts. Compared to the baseline bidirectional Dijkstra search, MEPW_C has a speed-up of more than 1 000 and even requires *less* space[6].

[5]The initial publication on binary edge restrictions [122] missed some optimizations, namely edge reduction, witness restrictions, the combination with ALT, and lower-bound pruning.

[6]This is possible as edges need to be stored only with the less important node.

Table 5.18: Experiments on NAV-NA showing combinations of results for (S)ingle-target witness search, (M)ulti-target witness search, (E)dge reduction, (P)arallel edge skipping, (W)itness restrictions (W_C for core-only witness restrictions for a core size of 10 000), and (G)oal-directed search for both single (G_1) and multiple (G_N) landmark sets with fixed core sizes of 10 000 (10k), 5 000 (5k), and 3 000 (3k) for both (U)ncontracted and (C)ontracted cores, with the option of (L)ower-bound pruning. All results are compared against the baseline bidirectional (D)ijkstra search (with no preprocessing).

	preprocessing		queries			
	time	space	time	settled	stalled	relaxed
algorithm	[hh:mm]	[B/node]	[ms]	nodes	nodes	edges
D	0:00	35	3 462.75	7 212 135	N/A	17 961 846
S	72:00+	-	-	-	-	-
M	16:05	33	7.29	1 033	624	49 436
ME	6:56	32	4.61	946	563	31 320
MEP	6:56	32	4.29	946	563	8 979
MEPW	7:15	41	3.36	834	450	6 769
$MEPW_C$	7:11	32	3.27	855	470	6 881
$MEPW_CG_1$10kU	0:49	32	6.88	3 806	135	51 437
$MEPW_CG_1$5kU	1:19	32	5.34	2 342	184	36 607
$MEPW_CG_1$3kU	1:56	32	4.51	1 744	227	28 377
$MEPW_CG_1$10kUL	0:49	32	5.35	2 715	133	28 536
$MEPW_CG_1$5kUL	1:19	32	4.22	1 794	182	20 754
$MEPW_CG_1$3kUL	1:56	32	3.66	1 410	223	16 191
$MEPW_CG_N$10kUL	0:56	37	4.38	1 971	133	19 428
$MEPW_CG_N$5kUL	1:26	34	3.87	1 440	182	15 706
$MEPW_CG_N$3kUL	2:02	33	3.42	1 192	223	12 803
$MEPW_CG_N$10kCL	7:21	37	1.18	491	133	3 073
$MEPW_CG_N$5kCL	7:20	34	1.50	578	180	3 675
$MEPW_CG_N$3kCL	7:20	33	1.75	648	221	4 072

We can significantly decrease the precomputation time when we do not contract core nodes. As a query would then settle almost all nodes in the core, we use ALT in the core to improve query performance. We can trade precomputation time for query time by choosing different core sizes. It is important to use lower-bound pruning, as this reduces query time especially on large cores by around 22%. The resulting algorithm $MEPW_CG_1$10kUL with a core size of 10 000 nodes has a preprocessing time of just 50 minutes, but the query time is more than 63% larger compared to $MEPW_C$. Leaving only 3 000 nodes uncontracted results in roughly 2 hours precomputation time, but increases the query time only by 12%. As we need to store the landmark distances only for core nodes, the space consumption does not visibly change.

We can further decrease the query time by using multiple landmark sets. This slightly

Table 5.19: Experiments on NAV-EU with the same settings as in Table 5.18.

	preprocessing		queries			
algorithm	time [hh:mm]	space [B/node]	time [ms]	settled nodes	stalled nodes	relaxed edges
D	0:00	33	6 273.21	11 366 174	N/A	25 842 792
S	72:00+	-	-	-	-	-
M	40:11	31	10.09	1 240	761	58 854
ME	17:27	30	6.60	1 222	704	42 815
MEP	17:27	30	6.05	1 222	704	15 301
MEPW	18:22	39	4.90	1 091	572	11 717
MEPW_C	18:10	30	4.52	1 119	601	11 897
$\text{MEPW}_\text{C}\text{G}_1$10kU	2:03	30	15.31	5 757	196	111 698
$\text{MEPW}_\text{C}\text{G}_1$5kU	3:36	30	12.99	3 726	287	87 378
$\text{MEPW}_\text{C}\text{G}_1$3kU	5:40	30	10.75	2 764	368	68 973
$\text{MEPW}_\text{C}\text{G}_1$10kUL	2:03	30	13.50	4 380	192	69 579
$\text{MEPW}_\text{C}\text{G}_1$5kUL	3:36	30	11.03	2 914	277	58 413
$\text{MEPW}_\text{C}\text{G}_1$3kUL	5:40	30	9.36	2 251	354	47 068
$\text{MEPW}_\text{C}\text{G}_N$10kUL	2:15	33	11.09	3 410	192	51 873
$\text{MEPW}_\text{C}\text{G}_N$5kUL	3:49	31	9.50	2 484	277	49 193
$\text{MEPW}_\text{C}\text{G}_N$3kUL	6:07	31	8.64	1 999	354	41 068
$\text{MEPW}_\text{C}\text{G}_N$10kCL	18:29	33	2.68	791	191	8 505
$\text{MEPW}_\text{C}\text{G}_N$5kCL	18:28	31	3.65	990	276	11 806
$\text{MEPW}_\text{C}\text{G}_N$3kCL	18:28	31	3.86	1 096	355	12 337

increases the precomputation time by about 6–7 minutes. The space consumption increases by 3% (for 3k core) to 16% (for 10k core). The resulting query time for the 3k core is now only 5% above MEPW_C.

The fastest query times are achieved using landmarks on a large contracted core. On a 10k core, our $\text{MEPW}_\text{C}\text{G}_N$10kCL algorithm achieves a query time of 1.18 ms, that is 2 900 times faster than bidirectional Dijkstra.

Table 5.19 provides the performance results on NAV-EU. We see a similar performance compared to NAV-NA, although the absolute numbers are larger as the network is larger. In general, landmarks do not decrease the query time as much as for NAV-NA. The speed-up of MEPW_C over Dijkstra's algorithm is around 1 300, while the best speed-up with landmarks is around 2 300.

Restricted Search. To assess the performance of our algorithm for different values of query constraint parameters, we measure query performance for different restriction *cardinalities* of the query constraint parameters $p = \langle p_1, p_2, \ldots, p_r \rangle$. We define the cardinality of p in this context as the number of $p_i \neq 0$. For uniformity with binary restrictions, the parameterized restrictions are either set to 0 or the maximum value. On NAV-NA, for

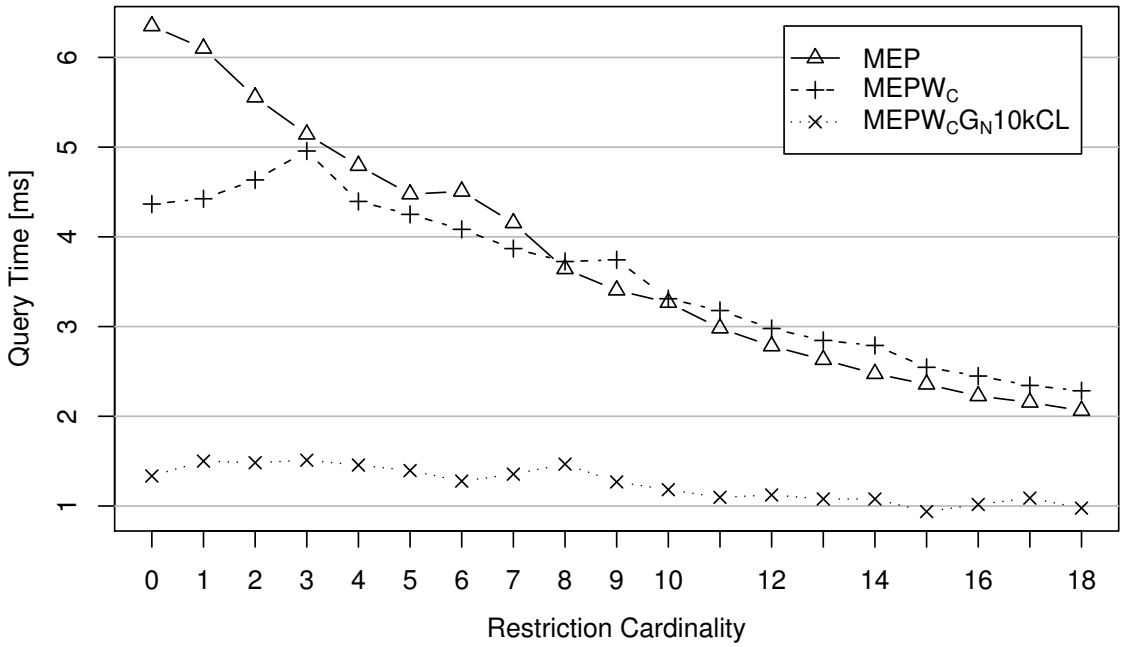


Figure 5.20: Experiments on NAV-NA comparing the query times of the MEP, MEPW$_C$, and MEPW$_C$G$_N$10kCL configurations across different restriction cardinalities.

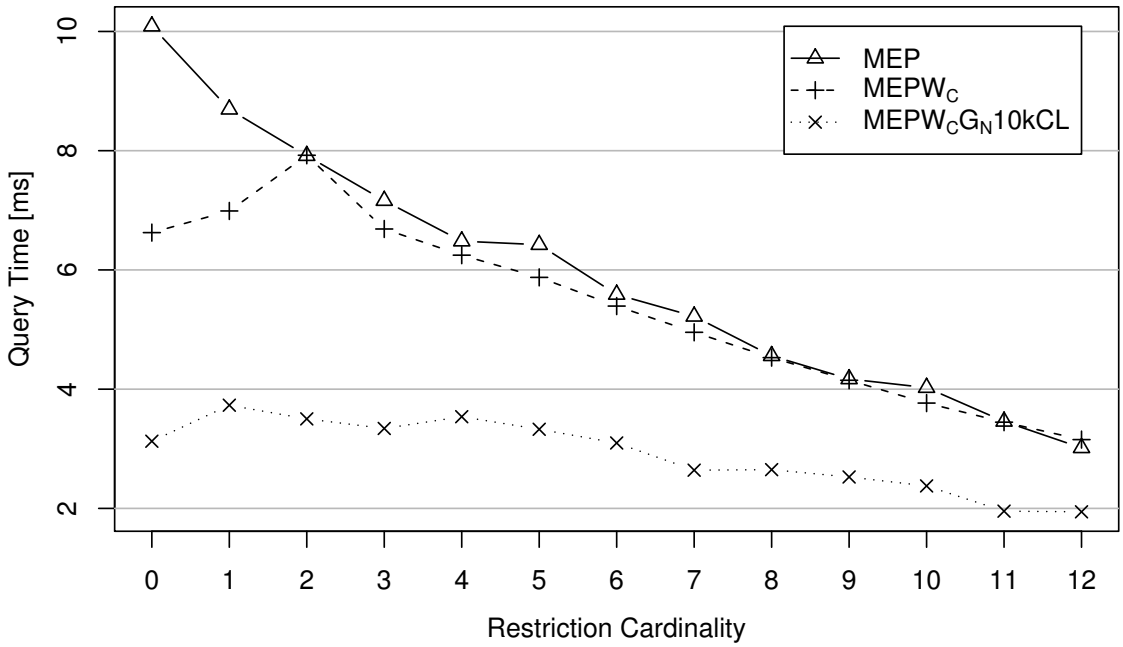


Figure 5.21: Experiments on NAV-EU comparing the query times of the MEP, MEPW$_C$, and MEPW$_C$G$_N$10kCL configurations across different restriction cardinalities.

each possible cardinality (0-18), we test the average query times of 10 000 source-target pairs, where each query is based on a uniform random set of query constraint parameters of the specified cardinality. We provide query times for three different algorithms in Figure 5.20. We see that without witness restrictions (MEP), the query time decreases with increasing restriction cardinality, as we need to relax fewer edges. As expected, witness restrictions significantly improve the performance of less restricted queries with small restriction cardinality up to 30%. For more restricted queries, MEP is slightly faster, but the overall performance is better for MEPW_C. Witness restrictions decrease the factor between unrestricted and restricted queries from 3.1 to 2.0. For $\text{MEPW}_\text{C}\text{G}_N$10kCL, our algorithm with the best query times, this factor is reduced even further to 1.4, providing very good performance independent of the restriction cardinality.

On NAV-EU, we observe similar results in Figure 5.21, where we have only twelve different restrictions. The absolute query times are larger, as the network is bigger. Witness restrictions improve the time of unrestricted queries by 34%.

Comparison to Static Contraction Hierarchies. In Table 5.22 we evaluate the performance of static (inflexible) contraction hierarchies for unrestricted queries ignoring any restrictions (IGNORE), and fully restricted queries having any restricted edge removed from the graph (REMOVE). We see that the preprocessing is much faster as it does not need to consider restrictions. IGNORE STATIC is 25 times faster than FLEXIBLE on NAV-NA, but we can still consider our preprocessing as efficient, as we consider $2^{16} \cdot 29 \cdot 57 = 108\,331\,008$ different choices of query constraint parameters, and on average, there are 29.4 different shortest paths over choices of query constraint parameters (Table 5.23). Also the query times of IGNORE STATIC are 11 times faster than FLEXIBLE, as the node order is tailored to the restriction set and not used for all possible restrictions. To evaluate this further, we perform the contraction with the node order computed by our flexible algorithm. The resulting query time is now only 4 times faster than FLEXIBLE, although the number of settled nodes and relaxed edges is almost the same. A big advantage of our flexible algorithm is its space-efficiency: its space consumption increases by less than 50%, mainly due to the larger edge data structure storing thresholds. This larger edge data structure also affects the cache-efficiency of our query algorithm, thus explaining some of its slowdown. For fully restricted queries (REMOVE), we see that the gap in the query time is much smaller, only a factor of 5. This is expected, as our flexible query algorithm can skip a lot of edges for such heavily-restricted queries. Also, it seems that the flexible node order is somewhat "closer" to the REMOVE node order, as using it only slightly increases the query time from 0.42 ms to 0.53 ms.

For NAV-EU, the gap between the static and our flexible algorithm is a bit larger. The preprocessing is now 39 times slower, although there are only 15.6 different shortest paths on average (Table 5.23). The graph seems to significantly change with restricted edges, indicated by the much higher number of edges carrying restrictions, cf. Table 5.17. One reason is possibly that, in Europe, many countries demand toll on all

Table 5.22: Static contraction results based on ignoring edge restrictions (IGNORE) and removing edges with restrictions (REMOVE), compared to our flexible algorithm (MEPW_C). We also give query performance in the case where we use the flexible node order (FLEX). In this case we report the flexible node ordering time (for MEP, this is the same node order as MEPW_C but does not compute witness restrictions) + the static contraction time. The performances are compared to our flexible algorithm (FLEXIBLE).

graph	query constraint parameters	algorithm	preprocessing time [hh:mm]	preprocessing space [B/node]	queries time [ms]	queries settled nodes	queries stalled nodes	queries relaxed edges
NAV-NA	IGNORE	STATIC	0:17	22	0.41	795	382	3 207
		STATIC (FLEX)	6:56 + 0:04	22	1.07	1 030	550	9 539
		FLEXIBLE	7:11	32	4.34	1 068	588	9 765
	REMOVE	STATIC	0:11	19	0.42	727	344	3 361
		STATIC (FLEX)	6:56 + 0:03	19	0.53	641	353	3 997
		FLEXIBLE	7:11	32	2.20	641	353	3 997
NAV-EU	IGNORE	STATIC	0:28	21	0.35	659	336	2 346
		STATIC (FLEX)	17:27 + 0:07	21	1.79	1 370	697	16 475
		FLEXIBLE	18:10	30	6.01	1 419	745	16 786
	REMOVE	STATIC	0:21	17	0.82	1 062	575	6 014
		STATIC (FLEX)	17:27 + 0:04	17	0.94	819	456	7 008
		FLEXIBLE	18:10	30	3.12	819	456	7 007

Table 5.23: Average and maximum number of unique shortest paths for individual s-t pairs over all possible query constraint parameters (when tested on 1000 s-t pairs).

graph	avg. path count	max. path count
NAV-NA	29.4	338
NAV-EU	15.6	135

their highways, whereas in North America only few highways require toll, mostly on the east coast of the United States. The results of Sanders and Schultes [130] suggest that restricting these highways highly affects the efficiency of our algorithms. Therefore, the balancing act of creating a node order suiting all restrictions is much harder, affecting the preprocessing and query time. Still, our algorithm is reasonably fast for web services, and more importantly, requires only about the space of two static contraction hierarchies.

Static Node Ordering. We are further interested in using statically computed node orders for our flexible contraction. In practice, the node ordering is much slower than contraction given a node order, as computing and updating the node priorities takes the majority of the time. Therefore, we hope that by performing a static node ordering, we can significantly decrease the overall precomputation time. The results are summarized

Table 5.24: Flexible contraction with restrictions, based on 2 static node orderings (from above): IGNORE and REMOVE. The preprocessing time is split into the static node ordering time + the flexible contraction time. We compare it to the flexible algorithm MEPW_C. Query results are based on averages between fully restricted and unrestricted queries.

		preprocessing		queries			
		time	space	time	settled	stalled	relaxed
graph	node order	[hh:mm]	[B/node]	[ms]	nodes	nodes	edges
NAV-NA	FLEXIBLE	7:11	30	3.27	855	470	6 881
	IGNORE	0:17 + 4:04	48	5.84	977	544	6 780
	REMOVE	0:11 + 48:00+	-	-	-	-	-
NAV-EU	FLEXIBLE	18:10	30	4.52	1 119	601	11 897
	IGNORE	0:28 + 39:21	45	22.09	1 426	930	20 215
	REMOVE	0:21 + 48:00+	-	-	-	-	-

in Table 5.24. We used the node orders IGNORE and REMOVE computed in the previous section. The IGNORE node order improves the precomputation time, as it only takes about 4 hours for the contraction plus an additional 17 minutes to compute the static node order, instead of 7:11. However, the query time increases by 79%. We analyzed this in more detail and found out that an unrestricted query takes now 2.53 ms and a fully restricted query takes 9.14 ms. So the static node order is only good for unrestricted queries, and is even faster than the flexible one for such queries. But restricted queries become slow, and the space consumption also increases by 60%, as a lot more shortcuts are necessary, since this static node order does not consider restrictions. We strongly see the blindness of a static node order towards unconsidered restrictions for the REMOVE node order. There, our contraction did not finish within two days. So we recommend the usage of static node orders only for people with profound knowledge of the algorithm and of their most commonly-expected query constraint scenarios, as a lot of things can go wrong.

5.4 Concluding Remarks

Review. Our algorithms are the first ones for fast flexible queries, and we still achieve query times comparable with static algorithms. We were able to engineer the preprocessing to be very time- and space-efficient. The resulting experiments for the flexible scenarios with multiple edge weights and edge restrictions show preprocessing times of several hours, and query times at the order of a few milliseconds. The speed-up over Dijkstra's algorithm is more than three orders of magnitude. We achieve this by incorporating an augmented version of ALT into our algorithm, and by further optimizations tailored to the specific flexible scenarios.

In the flexible scenario with two edges weights, our linear combination with a single parameter proved useful. It allows a very efficient node contraction. We improve the overall performance with the technique of parameter interval splitting, that adjusts the node ordering for different values of query parameters. Storing necessity intervals with edges helps to reduce the number of relaxed edges during query time. Finally, profile queries efficiently return all shortest paths within a value range of query parameters.

The flexible scenario with edge restrictions considers restrictions, such as bridge heights, that play an important role for the feasibility of a path. We develop witness restrictions, the equivalent of necessity intervals, that prune the relaxation of edges during less restricted queries. Using multiple sets of landmarks for different restrictions improves the performance. Also, this does not significantly increase the space overhead, as we store the landmark distances only for the most important nodes forming a core.

Future Work. We present two important flexible scenarios and developed efficient algorithms for each of them. However, in practice, they should be combined into a single algorithm. We provide the theoretical background to do so. But it is hard to estimate how many shortcuts will be necessary in such a combined scenario. Therefore, potentially more research is necessary to improve efficiency. A more advanced parameter splitting technique seems necessary to cope with the diverse possible values of query parameters. Such a parameter splitting technique can be used to easily parallelize and/or distribute the preprocessing. As the data stored with the edges becomes particularly large, even memory compression and external memory algorithms may be necessary if the available main memory is too small.

And, it is desirable to include further scenarios beyond the two scenarios tackled in this chapter. Among these are time-dependency and dynamization. Especially time-dependency imposes further challenges, as even the simple bi-criteria combination of time-dependent travel time and time-independent travel distance cannot be computed in forward direction but only in backward direction [75].

References. Section 5.2 is based on a conference paper [62], which the author of this thesis published together with Moritz Kobitzsch, and Peter Sanders, and on the diploma thesis of Moritz Kobitzsch [95]. Section 5.3 is based on a cooperation with Michael

Rice, Peter Sanders, and Vassilis Tsotras and the resulting paper [65] is currently under review by the Journal of Experimental Algorithms. Some wordings of these articles are used in this thesis.

6

Batched Shortest Paths Computation

6.1 Central Ideas

In this chapter, we consider shortest-path problems with multiple source-target pairs. The classical problem is the computation of a distance table: given a set of source nodes S and a set of target nodes T, we want to compute the shortest-path distances between *all* node pairs $(s,t) \in S \times T$. Computing these is important to solve routing problems for logistics, supply chain management and similar industries, but there are many more interesting applications.

In general, we have a set of queries $\Pi \subseteq V \times V$, and we need to compute the shortest-path distances between the pairs in Π. Define $S(\Pi) := \{s \mid (s,t) \in \Pi\}$ and $T(\Pi) := \{t \mid (s,t) \in \Pi\}$, and assume that $|S(\Pi)|$ and/or $|T(\Pi)|$ are much smaller than $|\Pi|$. There is a certain class of speed-up techniques that allows us to exploit this property. This class comprises of techniques that are *bidirected*, i. e. using a small forward search space

$$\overrightarrow{\sigma}(s) := \left\{ (u, \overrightarrow{\delta}(u)) \;\middle|\; u \text{ reached in forward search with distance } \overrightarrow{\delta}(u) \right\} \tag{6.1}$$

from the source node s and a small backward search space

$$\overleftarrow{\sigma}(t) := \left\{ (u, \overleftarrow{\delta}(u)) \;\middle|\; u \text{ reached in backward search with distance } \overleftarrow{\delta}(u) \right\} \tag{6.2}$$

from the target node t to find the shortest-path distance, and *non-goaldirected*, i. e. the small forward search does not depend on the target node and vice versa. The shortest-path distance is computed from the *intersection* of both search spaces, as these techniques ensure that

$$\mu(s,t) = \min_{u \in V} \left\{ \overrightarrow{\delta}(u) + \overleftarrow{\delta}(u) \;\middle|\; (u, \overrightarrow{\delta}(u)) \in \overrightarrow{\sigma}(s), (u, \overleftarrow{\delta}(u)) \in \overleftarrow{\sigma}(t) \right\} \quad . \tag{6.3}$$

If $|S(\Pi)|$ is much smaller than $|\Pi|$, we can compute $\overrightarrow{\sigma}(s)$ only once for each $s \in S(\Pi)$, and store it for repeated use. And symmetrically, if $|T(\Pi)|$ is much smaller than $|\Pi|$, we can compute $\overleftarrow{\sigma}(t)$ only once for each $t \in T(\Pi)$, and store it for repeated use. Then, for the computation of the shortest path distance $\mu(s,t)$, we only need to load $\overrightarrow{\sigma}(s)$ and $\overleftarrow{\sigma}(t)$ and *intersect* them.

6.1.1 Buckets

Instead of storing forward and/or backward search spaces with the origin of the search, it is sometimes more efficient to store them into buckets at the reached nodes [94]. More precisely, we store *forward buckets*

$$\overrightarrow{\beta}(u) := \left\{ (s, \overrightarrow{\delta}_s(u)) \mid (u, \overrightarrow{\delta}_s(u)) \in \overrightarrow{\sigma}(s), s \in S(\Pi) \right\} \tag{6.4}$$

and/or *backward buckets*

$$\overleftarrow{\beta}(u) := \left\{ (t, \overleftarrow{\delta}_t(u)) \mid (u, \overleftarrow{\delta}_t(u)) \in \overleftarrow{\sigma}(t), t \in T(\Pi) \right\} \quad . \tag{6.5}$$

This bucketing approach is especially helpful for online scenarios where, for example, target nodes rarely change, and the algorithm needs to answer shortest-path queries from an arbitrary source node to all of these target nodes immediately when the query arrives. In this case, we can store the backward search spaces from all target nodes in backward buckets. Then, upon the arrival of a query, we compute the forward search from the source node, and scan the backward buckets at the nodes reached by the forward search using

$$\mu(s,t) = \min \left\{ \overrightarrow{\delta}(u) + \overleftarrow{\delta}_t(u) \;\middle|\; (u, \overrightarrow{\delta}(u)) \in \overrightarrow{\sigma}(s), (t, \overleftarrow{\delta}_t(u)) \in \overleftarrow{\beta}(u) \right\} \quad . \tag{6.6}$$

Equation (6.6) is derived directly from (6.3) and (6.5).

In case that source nodes rarely change, a symmetric algorithm stores forward buckets, and computes the shortest-path distances using (6.7) deduced from (6.3) and (6.4).

$$\mu(s,t) = \min \left\{ \overrightarrow{\delta}_s(u) + \overleftarrow{\delta}(u) \;\middle|\; (u, \overleftarrow{\delta}(u)) \in \overleftarrow{\sigma}(t), (s, \overrightarrow{\delta}_s(u)) \in \overrightarrow{\beta}(u) \right\} \tag{6.7}$$

But not only online scenarios benefit from buckets, as scanning these buckets is very cache-efficient when their entries are stored in consecutive pieces of memory. Also note that the computation of the distances from a source node s to all target nodes $t \in T(\Pi)$ can be done simultaneously by maintaining an array of tentative distances, with one entry for each $t \in T(\Pi)$. Initially, the tentative distances are ∞, and we decrease them while scanning the buckets.

6.1.2 Further Optimizations

The ideas described so far allow to compute the shortest-path distances between all pairs in Π, but depending on the exact problem definition, additional algorithmic ingredients can speed the distance computation up:

- In the *time-dependent scenario*, travel times depend on the departure time, and edge weights are represented as travel time functions that map a departure time to a travel time. There, it pays of to compute additional data to speed up the intersection of $\overrightarrow{\sigma}(s)$ and $\overleftarrow{\sigma}(t)$.

- The *ridesharing problem* is to compute the offer with minimal detour to pickup a request. We precompute and store forward buckets from the start locations and backward buckets to the end locations for all offers. This allows to use them repeatedly to match against requests. Also by limiting the detour, we can prune the search and gain additional speed-up.
- To locate close *points-of-interest (POI)*, we precompute and store forward and backward buckets from the POI. Ordering the buckets by distance allows early pruning.

6.2 Time-dependent Travel Time Table Computation

A *travel time table* between a set of source nodes S and a set of target nodes T reduces the computation of minimum travel times to simple table lookups. In the time-dependent scenario (Section 2.2.2, [33, 54, 91]), these travel times depend on the departure time at the source node. Each cell in the travel time table corresponds to a travel time function (TTF) over the departure time. In contrast to the time-independent case, such a time-dependent table takes a lot longer to compute and occupies a lot more space. Therefore, we refine the problem of computing a table to the problem of implementing a *query interface*: Given $s \in S$ and $t \in T$, we want to know the earliest arrival time when we depart at time τ (or the travel time profile for all τ). So any algorithm that previously used a table now just needs to replace its table lookups with calls to this interface. An algorithm behind this interface uses a precomputed data structure with the knowledge of the graph, S and T to answer these queries fast. We contribute five such algorithms that allow different trade-offs between precomputation time and space and query time. Heuristic versions of these algorithms with approximation guarantees allow substantially faster and more space-efficient algorithms.

6.2.1 Preliminaries

The basic concepts of Chapter 3 can be augmented to the time-dependent scenario. In particular, we use time-dependent modifications of Dijkstra's algorithm (Section 3.1) and node contraction (Section 3.2). We further introduce approximate travel time functions, an important ingredient for optimizations.

Dijkstra's Algorithm. Given a departure time τ at the source node s, Dijkstra's algorithm is augmented by evaluating an edge (u,v) with the arrival time at node u [33]. This modified algorithm computes *earliest arrival times* at all nodes in the graph. The FIFO-property ensures the correctness of this modification.

To compute the travel time profiles from the source node s independent of the departure time, additional changes are necessary [116]. The modifications to this resulting *profile search* are as follows:

- *Node labels.* Each node v has a tentative TTF $\delta(v)$ from s to v.
- *Priority queue (PQ).* The keys used are the global minima of the labels. Reinserts into the PQ are possible and happen (*label correcting*).
- *Edge Relaxation.* Consider the relaxation of an edge (u,v). The label $\delta(v)$ of the node v is updated by computing the minimum TTF of $\delta(v)$ and $c(u,v) * \delta(u)$.

Node Contraction. The basic idea of node contraction remains the same in the time-dependent scenario: We contract nodes and add shortcuts to preserve shortest paths distances. However, we must ensure that these paths are preserved at *any time*. A witness search is therefore a profile search and we only can omit a shortcut if there is a witness that is not slower as the shortcut for any possible time τ. Note that we only need to perform node contraction once per graph independent of S and T. Time-dependent Contraction Hierarchies (TCH) [15, 16] is currently the most efficient time-dependent algorithm based on node contraction.

Furthermore, a query becomes more complex, as there are potentially several optimal candidate nodes. A profile query can be answered by performing a bidirectional upward profile search where both search scopes meet at *candidate* nodes u. The travel time profile is $\min\left\{\overleftarrow{\delta}(u) * \overrightarrow{\delta}(u) \mid u \text{ candidate}\right\}$.

Approximations. Approximate TTFs usually have fewer points and are therefore faster to process and require less memory.

Definition 6.1 *Let f be a TTF. A* lower bound *is a TTF $f^{\downarrow}$ with $f^{\downarrow} \leq f$ and a* lower ε-bound *if further $(1-\varepsilon)f \leq f^{\downarrow}$. An* upper bound *is a TTF $f^{\uparrow}$ with $f \leq f^{\uparrow}$ and an* upper ε-bound *if further $f^{\uparrow} \leq (1+\varepsilon)f$. An* ε-approximation *is a TTF $f^{\updownarrow}$ with $(1-\varepsilon)f \leq f^{\updownarrow} \leq (1+\varepsilon)f$.*

For example, the simplest lower/upper bound is $\tau \mapsto \min f / \max f$.

A profile search based on lower/upper bounds is a *bounded approximate search* [15]. It runs usually much faster than an exact profile search, whose tentative TTFs contain a lot of points. Essentially, the bounded approximate search is two profile searches, one based on lower bounds of the edge TTFs and one based on upper bounds of the edge TTFs. But in practice, both searches are executed together. The results are a lower and an upper bound on the travel time profile. The fastest variant of bounded approximate search is based on the global minima and maxima of the edge TTFs, we call it *min-max search*.

6.2.2 Five Algorithms

We engineer five algorithms with different precomputation time, space and query time to implement the time query interface (s,t,τ) which computes the earliest arrival time at $t \in T$ when we depart at $s \in S$ at time τ, and the profile query interface (s,t) which computes the travel time profile between $s \in S$ and $t \in T$. Our algorithms have in common that they need to precompute $\forall s \in S$ the forward search spaces

$$\overrightarrow{\sigma}(s) := \left\{ (u, \overrightarrow{\delta}(u)) \mid \overrightarrow{\delta}(u) \text{ is TTF from } s \text{ to } u \text{ in forward upward search} \right\}$$

and $\forall t \in T$ the symmetric backward search spaces

$$\overleftarrow{\sigma}(s) := \left\{ (u, \overleftarrow{\delta}(u)) \mid \overrightarrow{\delta}(u) \text{ is TTF from } u \text{ to } t \text{ in backward upward search} \right\} .$$

Note that in comparison to the static table computation algorithm [94], we need to store a TTF not at the candidate node u, but at the source or target node, as this is necessary to perform the intersection. The algorithms are ordered by decreasing query time.

Algorithm 6.1: IntersectTimeQuery(s,t,τ)

input : source node s, target node t, departure time τ
output : earliest arrival time

```
1 δ := ∞;                                                  // tentative arrival time
2 foreach (u, δ→(u)) ∈ σ→(s), (u, δ←(u)) ∈ σ←(t) do          // candidate nodes
3     if τ + min δ→(u) + min δ←(u) < δ then                 // prune using minimum
4         δ' := δ→(u)(τ) + τ;                               // evaluate TTFs
5         δ' := δ←(u)(δ') + δ';
6         δ := min(δ, δ');                                  // update tentative arrival time
7 return δ
```

Algorithm INTERSECT computes and stores $\overrightarrow{\sigma}(s)$ and $\overleftarrow{\sigma}(t)$. Algorithm 6.1 shows the *time query* computing the earliest arrival time. The main part is to evaluate all paths via the candidate nodes. At most 2·#candidates TTF evaluations are required for a query. However, the TTF evaluations are the most expensive part, so we prune them using the (precomputed) minima of $\overrightarrow{\delta}(u)$, $\overleftarrow{\delta}(u)$.

A *profile query* computing the travel time profile between s and t is similar to a time query, but links the two TTFs at the candidate instead of evaluating them. But as the link operation is even more expensive as the evaluation operation, we implement more sophisticated pruning steps, see Algorithm 6.2. During the computation of the search spaces, for each $(u, \overrightarrow{\delta}(u)) \in \overrightarrow{\sigma}(s)$ we compute and store lower/upper ε-bounds $\overrightarrow{\delta}^{\downarrow}(u) \Big/ \overrightarrow{\delta}^{\uparrow}(u)$ and for each $(u, \overleftarrow{\delta}(u)) \in \overleftarrow{\sigma}(t)$ we compute and store lower/upper ε-bounds $\overleftarrow{\delta}^{\downarrow}(u) \Big/ \overleftarrow{\delta}^{\uparrow}(u)$. Then we pass three times through the search spaces $\overrightarrow{\sigma}(s)$, $\overleftarrow{\sigma}(t)$:

Algorithm 6.2: IntersectProfileQuery(s,t)

input : source node s, target node t
output : travel time profile

```
1  δ̄ := ∞;                                              // upper bound based on maxima
2  δ̲ := ∞;                                              // lower bound based on minima
3  u̲ := ⊥;                                              // minimum candidate
4  foreach (u,·) ∈ σ⃗(s), (u,·) ∈ σ⃖(t) do               // candidate nodes
5      if max δ⃗(u) + max δ⃖(u) < δ̄ then
6          δ̄ := max δ⃗(u) + max δ⃖(u);                   // update upper bound
7      if min δ⃗(u) + min δ⃖(u) < δ̲ then
8          δ̲ := min δ⃗(u) + min δ⃖(u);                   // update lower bound
9          u̲ = u;                                       // update minimum candidate
10 δ↑ := δ⃖↑(u̲) ∗ δ⃗↑(u̲);                              // upper bound based on approximate TTFs
11 δ̄ := min(δ̄, max δ↑);                                // tighten upper bound
12 foreach (u,·) ∈ σ⃗(s), (u,·) ∈ σ⃖(t) do               // candidate nodes
13     if min δ⃗(u) + min δ⃖(u) ≤ δ̄ then                 // prune using minima
14         δ↑ := min(δ↑, δ⃖↑(u) ∗ δ⃗↑(u));               // update upper bound
15 δ := δ⃖(u̲) ∗ δ⃗(u̲);                                  // tentative travel time profile
16 foreach (u, δ⃗(u)) ∈ σ⃗(s), (u, δ⃖(u)) ∈ σ⃖(t) do       // candidate nodes
17     if ¬(δ⃖↓(u) ∗ δ⃗↓(u) > δ↑) then                   // prune using lower bounds
18         δ := min(δ, δ⃖(u) ∗ δ⃗(u));                   // update travel time profile
19 return δ
```

1. (Lines 1–9) First, we compute an upper bound $\overline{\delta}$ based on the maxima of the search space TTFs. In the same pass, we compute a candidate $\underline{u}$ with minimum sum of the minima of the search space TTFs. This candidate is usually very important and a good starting point to obtain a tight lower bound.

2. (Lines 10–14) Then, we compute a tighter upper bound $\delta^{\uparrow}$ based on the minimum over all upper ε-bounds. We use $\overline{\delta}$ for pruning, and additionally initialize $\delta^{\uparrow}$ with the upper ε-bound via candidate $\underline{u}$.

3. (Lines 15–18) Finally, we compute the travel time profile. We initialize the tentative travel time profile with the TTF via candidate $\underline{u}$. We prune each candidate whose lower ε-bound is larger than the computed upper bound $\delta^{\uparrow}$ for all departure times. So we execute the very expensive link and minima operations (Line 18) only at a few relevant candidates.

An important observation is that the computation time and space of a single search

space depend only on the graph, and are *independent* of $|S|$ and $|T|$. INTERSECT requires therefore $\Theta(|S|+|T|)$ preprocessing time and space, and $\Theta(1)$ query time, if only $|S|$ and $|T|$ are considered as changing variables.

Algorithm MINCANDIDATE precomputes and stores the *minimum* candidate in a table $c_{\min}(s,t)$.

$$c_{\min}(s,t) := \operatorname*{argmin}_{u \text{ candidate}} \left\{ \min \overrightarrow{\delta}(u) + \min \overleftarrow{\delta}(u) \right\}$$

By that, we can use it to obtain a good initial upper bound for a time query, by initializing δ in Line 1 of Algorithm 6.1 with the travel time via $c_{\min}(s,t)$. This allows to prune more candidates and results in faster query times. However, preprocessing time and space are now in $\Theta(|S| \cdot |T|)$, but with a very small constant factor.

Algorithm RELEVANTCANDIDATES precomputes a set of candidate nodes $c_{\text{rel}}(s,t)$ for each s-t-pair by using lower and upper bounds on the TTFs in $\overrightarrow{\sigma}(s)$ and $\overleftarrow{\sigma}(t)$.

$$c_{\text{rel}}(s,t) := \left\{ u \;\middle|\; \neg \left(\overleftarrow{\delta}^{\downarrow}(u) * \overrightarrow{\delta}^{\downarrow}(u) > \min_{v \text{ candidate}} \left\{ \overleftarrow{\delta}^{\uparrow}(v) * \overrightarrow{\delta}^{\uparrow}(v) \right\} \right) \right\}$$

This is exactly the set of nodes that are evaluated in Line 18 of Algorithm 6.2. So it is sufficient to evaluate the candidates in $c_{\text{rel}}(s,t)$ to answer a query correctly. In practice, $c_{\text{rel}}(s,t)$ is stored as an array with $c_{\min}(s,t)$ on the first position. Additionally, we can save space by not storing $(u, \overrightarrow{\delta}(u))$ in $\overrightarrow{\sigma}(s)$ if $\forall t \in T : u \notin c_{\text{rel}}(s,t)$, and symmetrically for $\overleftarrow{\sigma}(t)$. The precomputation time depends on the used lower and upper bounds: Using only min-max-values is fast but results in larger sets, using ε-bounds is slower but reduces the size of the sets.

Algorithm OPTCANDIDATE precomputes for every departure time τ an *optimal* candidate $c_{\text{opt}}(s,t,\tau)$, so a time query only needs to evaluate one candidate.

$$c_{\text{opt}}(s,t,\tau) := \operatorname*{argmin}_{u \text{ candidate}} \left\{ \left(\overleftarrow{\delta}(u) * \overrightarrow{\delta}(u) \right)(\tau) \right\}$$

A candidate is usually optimal for a whole period of time, we store these periods as consecutive, non-overlapping, intervals $[\tau_1, \tau_2)$. In practice, there are only very few intervals per pair (s,t) so that we can find the optimal candidate very fast. The downside of this algorithm is its very high precomputation time since it requires the computation of the travel time profile for each pair $(s,t) \in S \times T$ with the INTERSECT algorithm. Still, storing only the optimal candidates requires usually less space than the travel time profile.

Algorithm TABLE computes and stores all travel time profiles in a table.

$$\text{table}(s,t) := \min_{u \text{ candidate}} \left\{ \overleftarrow{\delta}(u) * \overrightarrow{\delta}(u) \right\}$$

It provides the fastest query times, but the space requirements in $\Theta(|S| \cdot |T|)$ have a large constant factor. The table cells are computed with the INTERSECT algorithm.

Correctness. We will only prove the correctness of the INTERSECT algorithm in Lemmas 6.2 and 6.3. The correctness of the other four algorithms follows, as they essentially do the same, but move some computation to the preprocessing.

Lemma 6.2 *Algorithm 6.1 returns* $\mu(s,t,\tau)$.

Proof. Under the assumption that no pruning would happen, the algorithm would be correct due to the correctness of a TCH query. If we prune candidate u (Line 3), then $\tau + \min \overrightarrow{\delta}(u) + \min \overleftarrow{\delta}(u) \geq \delta \Rightarrow \tau + \overrightarrow{\delta}(u)(\tau) + \overleftarrow{\delta}(u)(\tau + \overrightarrow{\delta}(u)) \geq \delta$. So candidate u would not decrease δ. □

Lemma 6.3 *Algorithm 6.2 returns* $\mu(s,t)$.

Proof. Under the assumption that no pruning would happen, the algorithm would be correct due to the correctness of a TCH query. Naturally, $\delta^{\uparrow}$ is an upper bound on $\mu(s,t)$, the pruning in Line 13 could only potentially worsen the tightness of the upper bound. If we prune candidate u based on $\delta^{\uparrow}$ (Line 17), then $\mu(s,t) \leq \delta^{\uparrow} < \overleftarrow{\delta}(u) * \overrightarrow{\delta}(u)$. So candidate u would not decrease δ for any time τ. □

6.2.3 Computation of Search Spaces

The computation of the target-independent forward profile search spaces $\overrightarrow{\sigma}(s)$ for $s \in S$, and the source-independent backward profile search spaces $\overleftarrow{\sigma}(t)$ for $t \in T$ is done using a suitable hierarchical speed-up technique. We use TCH [16], and will now describe in detail how to efficiently compute the search spaces. The simplest way to compute them is to use upward profile searches as described in Section 6.2.1 To reduce the computational effort, we initially perform a min-max search using the *stall-on-demand* technique and use it to prune the following profile search. This technique *stalls* nodes that are reached suboptimally. Note that a node can be reached suboptimally, as our upward search does not relax *downward* edges of a settled node. These stalled nodes will never be a candidate to an shortest path, as they are reached suboptimally. Furthermore, we prune an edge (u,v), if the minimum duration computed for u plus the minimum duration of the edge is larger than the maximum duration computed for v. Finally, we perform a profile search to compute the search space only using nodes that are not stalled and edges that are not pruned. The conditions under which nodes are stalled and edges are pruned ensure that

the correctness of our algorithms is not affected. Also, stalling does not only speed-up the computation of the search spaces $\overrightarrow{\sigma}(s)$, $\overleftarrow{\sigma}(t)$, but also reduces the size of them, as we do not need to store stalled nodes.

Note that the point-to-point TCH algorithm [16] applies further pruning techniques before the final profile search is executed. These techniques are usable once an upper bound on the whole shortest-path distance is obtained. However, as we want the search spaces only to be dependent on source or target node, but not on both, we cannot use these further pruning techniques.

6.2.4 Approximate Travel Time Functions

Approximate TTFs [16] reduce preprocessing time, space and query time of the algorithms in the previous section by several orders of magnitude. We consider three places to approximate TTFs: the TTFs assigned to the edges of the TCH, the node label TTFs after the computation of the forward/backward searches and finally the TABLE entries. Approximating the latter two can be applied straightforwardly. But replacing the exact edge TTFs in the TCH with approximate ones requires some caution.

To ensure that the computation of the search spaces (Section 6.2.3) works as expected, we need to modify the stall-on-demand technique. This is necessary, as the TCH was constructed with exact TTFs.[1] As we use the stall-on-demand technique only during min-max search, we perform the min-max search on the min-max values of the exact TTFs. The latter profile query is then performed on the approximate TTFs.

The query algorithms stay the same, except that INTERSECT profile queries no longer use ε-bounds for pruning, as the overhead does no longer pay off.

Lemmas 6.4–6.6 enable us to compute theoretical error bounds. Note that we are the first to provide these theoretical error bounds and they can also be applied to the approximate point-to-point TCH algorithm [16]. With Lemma 6.6 we have an error bound on the search space TTFs when we approximate the edge TTFs with ε_e:

$$\varepsilon_1 := \varepsilon_e(1+\alpha)/(1-\alpha\varepsilon_e)$$

The error bound for approximating the search space TTFs with ε_s follows directly from Definition 6.1 of an ε_s-approximation:

$$\varepsilon_2 := (1+\varepsilon_s)(1+\varepsilon_1) - 1$$

Lemma 6.4 gives an error bound when we link the forward and backward search TTF on a candidate node:

$$\varepsilon_3 := \varepsilon_2(1+(1+\varepsilon_2)\alpha)$$

With Lemma 6.5 we know that ε_3 is an error bound on the approximate travel time profile, the minimum over all candidate TTFs. When we additionally approximate the

[1] Note that approximating the TTFs on the original graphs makes usually no sense, as these TTFs are already very simple.

resulting profile TTF for the table with ε_t, the error bound follows directly from Definition 6.1 of an ε_t-approximation:

$$\varepsilon_4 := (1+\varepsilon_t)(1+\varepsilon_3) - 1$$

In Table 6.18 we compute the resulting error bounds for our test instances.

Lemma 6.4 *Let $f^\updownarrow$ be an ε_f-approximation of TTF f and $g^\updownarrow$ be an ε_g-approximation of TTF g. Let α be the maximum slope of g, i. e. $\forall \tau' > \tau : g(\tau') - g(\tau) \leq \alpha(\tau' - \tau)$. Then $g^\updownarrow * f^\updownarrow$ is a $\max\left\{\varepsilon_g, \varepsilon_f(1+(1+\varepsilon_g)\alpha)\right\}$-approximation of $g * f$.*

Proof. Let τ be a time.

$$\begin{aligned}
(g^\updownarrow * f^\updownarrow)(\tau) &= g^\updownarrow(f^\updownarrow(\tau)+\tau) + f^\updownarrow(\tau) \\
&\leq (1+\varepsilon_g)(g(f^\updownarrow(\tau)+\tau)) + (1+\varepsilon_f)f(\tau) \\
&\leq (1+\varepsilon_g)(g(f(\tau)+\tau) \\
&\quad +\alpha\,|(f^\updownarrow(\tau)+\tau) - (f(\tau)+\tau)|) + (1+\varepsilon_f)f(\tau) \\
&\leq (1+\varepsilon_g)(g(f(\tau)+\tau) + \alpha\varepsilon_f f(\tau)) + (1+\varepsilon_f)f(\tau) \\
&= (1+\varepsilon_g)g(f(\tau)+\tau) + (1+\varepsilon_f(1+(1+\varepsilon_g)\alpha))f(\tau)
\end{aligned}$$

By applying the symmetric transformations, we also obtain $(g^\updownarrow * f^\updownarrow)(\tau) \geq (1-\varepsilon_g)g(f(\tau)+\tau) + (1-\varepsilon_f(1+(1-\varepsilon_g)\alpha))f(\tau)$. □

Lemma 6.5 *Let $f^\updownarrow$ be an ε_f-approximation of TTF f and $g^\updownarrow$ be an ε_g-approximation of TTF g. Then $\min(f^\updownarrow, g^\updownarrow)$ is a $\max\left\{\varepsilon_f, \varepsilon_g\right\}$-approximation of $\min(f,g)$.*

Proof. Let τ be a time, WLOG we assume that $\min(f^\updownarrow, g^\updownarrow)(\tau) = f^\updownarrow(\tau)$ and $\min(f,g)(\tau) = g(\tau)$. Then $\min(f^\updownarrow, g^\updownarrow)(\tau) = f^\updownarrow(\tau) \leq g^\updownarrow(\tau) \leq (1+\varepsilon_g)g(\tau)$ and $\min(f^\updownarrow, g^\updownarrow)(\tau) = f^\updownarrow(\tau) \geq (1-\varepsilon_f)f(\tau) \geq (1-\varepsilon_f)g(\tau)$. □

Lemma 6.6 *Let $\overrightarrow{\sigma}(s)$ and $\overleftarrow{\sigma}(t)$ be the forward/backward search spaces computed on a TCH and $\overrightarrow{\sigma}^\updownarrow(s)$ and $\overleftarrow{\sigma}^\updownarrow(t)$ on the same TCH with ε-approximated edge TTFs. In both cases, stall-on-demand was only used with exact min-max values. Let α be the maximum slope of all TTFs in $\overrightarrow{\sigma}(s)$, $\overleftarrow{\sigma}(t)$, and $\alpha\varepsilon < 1$. Then $\left\{u \;\middle|\; (u, \overrightarrow{\delta}(u)) \in \overrightarrow{\sigma}(s)\right\} = \left\{u \;\middle|\; (u, \overrightarrow{\delta}^\updownarrow(u)) \in \overrightarrow{\sigma}^\updownarrow(s)\right\}$ and $\overrightarrow{\delta}^\updownarrow(u)$ is an $\varepsilon(1+\alpha)/(1-\alpha\varepsilon)$-approximation of $\overrightarrow{\delta}(u)$, and the same holds for the backward search spaces.*

Proof. $\left\{u \;\middle|\; (u,\cdot) \in \overrightarrow{\sigma}(s)\right\} = \left\{u \;\middle|\; (u,\cdot) \in \overrightarrow{\sigma}^\updownarrow(s)\right\}$ holds trivially since exact and approximate search both use the same min-max values, the same for the backward search spaces. WLOG we assume that the nodes are numbered $1..n$ by order of contraction, node 1 being contracted first. For the forward search, we will prove via induction over s (starting with the most important node n) that for each $\overrightarrow{\delta}^\updownarrow(u)$ in $\overrightarrow{\sigma}^\updownarrow(s)$ there exists a $k \in \mathbb{N}$ so that $\overrightarrow{\delta}^\updownarrow(u)$ is a $((1+\varepsilon)\left(\sum_{i=0}^{k-1}(\alpha\varepsilon)^i\right) - 1)$-approximation of $\overrightarrow{\delta}(u)$.

The base case holds trivially since for $s = n$, $\overrightarrow{\delta}(n) = \{(n,0)\} = \overrightarrow{\sigma}^\updownarrow(n)$.

Inductive step: Let $(u, \overrightarrow{\delta}(u)) \in \overrightarrow{\sigma}(s)$, $(u, \overrightarrow{\delta}^{\updownarrow}(u)) \in \overrightarrow{\sigma}^{\updownarrow}(s)$, and $N := \{v \mid (s,v) \in E, s < v\}$. Further let $c(s,v)$ be the exact TTF on the edge (s,v) and $c^{\updownarrow}(s,v)$ its ε-approximation. By definition of N,

$$\overrightarrow{\delta}(u) = \min\left\{ \overrightarrow{\delta_v}(u) * c(s,v) \;\middle|\; v \in N, (u, \overrightarrow{\delta_v}(u)) \in \overrightarrow{\sigma}(v) \right\}$$

and

$$\overrightarrow{\delta}^{\updownarrow}(u) = \min\left\{ \overrightarrow{\delta_v}^{\updownarrow}(u) * c^{\updownarrow}(s,v) \;\middle|\; v \in N, (u, \overrightarrow{\delta_v}^{\updownarrow}(u)) \in \overrightarrow{\sigma}^{\updownarrow}(v) \right\} \quad .$$

By induction hypothesis there exists $k \in \mathbb{N}$ so that $\overrightarrow{\delta_v}^{\updownarrow}(u)$ is an $((1+\varepsilon)\left(\sum_{i=0}^{k-1}(\alpha\varepsilon)^i\right) - 1)$-approximation of $\overrightarrow{\delta_v}(u)$. Also $\overrightarrow{\delta_v}(u)$ has maximum slope α and the edge TTF $c^{\updownarrow}(s,v)$ is an ε-approximation of $c(s,v)$. So by Lemma 6.4, $\overrightarrow{\delta_v}^{\updownarrow}(u) * c^{\updownarrow}(s,v)$ is a $((1+\varepsilon)\left(\sum_{i=0}^{k}(\alpha\varepsilon)^i\right) - 1)$-approximation of $\overrightarrow{\delta_v}(u) * c(s,v)$ ($k \rightsquigarrow k+1$).

Lemma 6.5 finally shows that the induction hypothesis holds for $\overrightarrow{\delta}^{\updownarrow}(u)$. So for any TTF in any $\overrightarrow{\sigma}^{\updownarrow}(s)$ there exists this $k \in \mathbb{N}$, and with $\alpha\varepsilon < 1$ we follow $\lim_{k\to\infty}((1+\varepsilon)\left(\sum_{i=0}^{k-1}(\alpha\varepsilon)^i\right) - 1) = \varepsilon(1+\alpha)/(1-\alpha\varepsilon)$. This concludes the proof for the forward case. The backward case is similar, except that the induction is over node u with $(u,\cdot) \in \overleftarrow{\sigma}(t)$, starts with the least important node, and uses $N := \{v \mid (u,v) \in E, v < u\}$,

$$\overleftarrow{\delta}(u) = \min\left\{ \overleftarrow{\delta}(v) * c(u,v) \mid v \in N, (v, \overleftarrow{\delta}(v)) \in \overleftarrow{\sigma}(t) \right\}$$

and

$$\overleftarrow{\delta}^{\updownarrow}(u) = \min\left\{ \overleftarrow{\delta}^{\updownarrow}(v) * c^{\updownarrow}(u,v) \;\middle|\; v \in N, (v, \overleftarrow{\delta}^{\updownarrow}(v)) \in \overleftarrow{\sigma}^{\updownarrow}(t) \right\} \quad .$$

□

6.2.5 On Demand Precomputation

We discussed five algorithms with different precomputation times in Section 6.2.2. Only the first algorithm INTERSECT provides precomputation in $\Theta(|S|+|T|)$. All further algorithms are in $\Theta(|S|\cdot|T|)$ as they precompute some data for each pair in $S \times T$. To provide a linear algorithm that benefits from the ideas of the further algorithms, we can compute the additional data ($c_{\min}(s,t)$, $c_{\text{rel}}(s,t)$, $c_{\text{opt}}(s,t,\tau)$ or table(s,t)) on demand only for those pairs (s,t) that occur in queries. By that, our algorithm is in $\Theta(|S|+|T|+\#\text{queries})$.

While a profile query already computes all the additional data at the first occurrence of (s,t), a time query does not. Depending on the additional precomputation time, we should only compute the additional data after a certain number of time queries for that pair occurred to improve the *competitive ratio*. This ratio is the largest possible ratio between the runtime of an algorithm that knows all queries in advance and our algorithm that does not (*online algorithm*). For just two different algorithms, e. g. INTERSECT and TABLE, this problem is similar to the ski-rental problem [86]. Let it "cost" t_i to

answer a query using INTERSECT and t_t using TABLE, and t_c to compute the table cell. Then computing the table cell on the query number $b = \lfloor t_c/(t_i - t_t) \rfloor$ to this cell has a competitive ratio < 2. In practice, we can predict the cost of t_i and t_t from the number of necessary TTF evaluations, and the cost t_c from the sum of the points of the TTFs $\overrightarrow{\delta}(u)$ and $\overleftarrow{\delta}(u)$ of all (relevant) candidates u.

When we want to use more than two of the five algorithms online, Azar et al. [8] propose an algorithm with competitive ratio 6.83. Note that although it has a worse competitive ratio, it may be still better in practice, as we can use more than two algorithms. The algorithm distinguishes between preprocessing time and query time. It needs to decide when to perform precomputation, and which algorithm to use on a cell. Let x and y be positive constants satisfying $2/x \leq 1$ and $1/x + 2y \leq 1$. Assume that we spent t_c time on the last precomputation, t_q on answering queries since the last precomputation, and T_q on answering queries in total for this cell. We will perform another precomputation before the next query if and only if $t_q \geq y \cdot t_c$ and there is an algorithm available with faster query times that requires at most $x \cdot T_q$ precomputation time. Among these algorithms we pick the one with the fastest query time. The competitive ratio of this algorithm is $1 + x + 1/y$. This term is smallest under the above constraints for $x = 1 + \sqrt{2}$ and $y = 1/(2 + \sqrt{2})$, resulting in a competitive ratio of $4 + 2\sqrt{2} \approx 6.83$.

6.2.6 Experiments

Instances. Our main instance is a real-world time-dependent road network of Germany (TD-GER) with 4.7 million nodes and 10.8 million edges, provided by PTV AG for scientific use. It reflects the midweek (Tuesday till Thursday) traffic collected from historical data, i. e., a high traffic scenario with about 8 % time dependent edges. Furthermore, we verified the robustness of our algorithms on two other instances. The first one is the same network of Germany, but now with Sunday traffic instead of midweek (TD-GER-SUN). The second instance is a network of Western Europe (TD-EUR) with about 18 million nodes and 42.6 million edges. It has been augmented with synthetic time-dependent travel times using a high amount of traffic where all but local and rural roads have time-dependent edge weights [112]. As there are many test results even for a single instance, we will first discuss our algorithms in depth regarding our main instance. Then, we will provide experimental results on our other two instances, and focus the discussion on the robustness of our algorithms.

Environment. Our machine has two Intel Xeon X5550 processors (Quad-Core) clocked at 2.67 GHz with 48 GiB of RAM and 2×8 MiB of Cache running SuSE Linux 11.1. Our C++ code was compiled by GCC 4.3.2 using optimization level 3.

Basic setup. We choose $S, T \subseteq V$ uniformly at random for a *size* $|S| = |T|$. We approximated the TTFs in the graph with ε_e, the TTFs of the search spaces with ε_s and the TTFs

in the table with ε_t. We use lower and upper ε_p-bounds for pruning profile queries, or just min-max-values if no ε_p is specified. The precomputation uses all 8 cores of our machine since it can be trivially parallelized and multi-core CPUs are standard these days. We report the *preprocessing* time to compute the forward and backward search spaces as *search* and the time to compute additional data ($c_{\min}$, c_{rel}, c_{opt} or table) as *link*. We also give the used *RAM* reported by the Linux kernel. The time (profile) query performances are averages over 100 000 (1 000) queries selected uniformly at random and performed on a single core. Depending on the algorithm, we also report some more detailed time query statistics. *Scan* is the number of nodes in the search spaces we scanned during a time query. *Meet* is the number of candidate nodes where forward and backward search space met. *Eval* is the number of TTF evaluations. *Succ* is the number of successful reductions of the tentative earliest arrival time due to another evaluated candidate.

The memory consumption of the input TCH is given in Table 6.1. It decreases significantly when we use approximate edge TTFs.

Table 6.1: Memory consumption of the TCH for different edge approximations.

	ε_e [%]	-	0.1	1.0	10.0
TD-GER	[MiB]	4 497	1 324	1 002	551
TD-GER-SUN	[MiB]	1 340	641	531	412
TD-EUR	[MiB]	10 175	4 132	3 387	2 516

Preprocessing time and search space size of the INTERSECT algorithm are in $\Theta(|S|+|T|)$ as expected, see Table 6.2. Note that the RAM requirements include the

Table 6.2: Performance of the INTERSECT algorithm.

				preprocessing		search spaces			query					
size	ε_e	ε_s	ε_p	search	RAM	[MiB]	TTF	point	time	scan	meet	eval	succ	profile
	[%]	[%]	[%]	[s]	[MiB]		#	#	[μs]	#	#	#	#	[μs]
100	-	-	0.1	7.5	6 506	1 639	172	2 757	5.17	310	19.6	9.97	3.92	1 329
500	-	-	0.1	33.8	13 115	8 228	172	2 768	7.43	312	20.0	10.38	4.08	1 494
1 000	-	-	0.1	68.0	21 358	16 454	173	2 754	7.97	313	19.9	10.28	4.04	1 412
1 000	-	-	-	53.1	20 830	15 897	173	2 754	7.99	313	19.9	10.28	4.04	7 633
1 000	1.0	-	-	1.5	1 579	349	173	54.4	6.13	313	19.9	10.27	4.05	108.2
1 000	-	1.0	-	64.9	5 302	72	173	6.3	6.46	313	19.9	10.60	4.05	18.4
1 000	0.1	0.1	-	4.7	1 749	189	173	26.7	6.29	313	19.9	10.32	4.04	52.8
1 000	1.0	1.0	-	1.8	1 303	65	173	5.1	5.48	313	19.9	10.36	4.05	15.1
1 000	10.0	10.0	-	0.7	854	47	173	1.9	6.34	313	19.9	15.79	4.05	22.0
10 000	1.0	1.0	-	18.2	2 015	650	174	5.1	6.80	315	19.9	10.34	4.01	16.3

Table 6.3: Performance of the MINCANDIDATE algorithm.

			preprocessing				search	query					
size	ε_e	ε_s	search	link	RAM	c_{min}	space	time	scan	meet	eval	succ	profile
	[%]	[%]	[s]	[s]	[MiB]	[MiB]	[MiB]	[μs]	#	#	#	#	[μs]
100	-	-	6.0	0.0	6 481	1	1 583	3.11	310	19.6	3.65	1.04	6 941
1 000	-	-	53.1	0.4	20 849	7	15 897	4.97	313	19.9	3.72	1.05	7 087
1 000	1.0	1.0	1.8	0.4	1 310	7	65	4.09	313	19.9	3.77	1.07	13.8
10 000	1.0	1.0	18.2	49.0	2 777	649	650	4.94	315	19.9	3.81	1.07	14.4

input TCH. The exact time query is two orders of magnitude faster than a standard TCH[2] time query (720 μs). However, the TCH profile query (32.75 ms) is just 22 times slower since most time is spent on computing the large resulting TTFs. Approximating the edge TTFs ($\varepsilon_e > 0$) reduces preprocessing time and RAM, approximating search spaces ($\varepsilon_s > 0$) reduces search space sizes. When we combine both to $\varepsilon_e = \varepsilon_s = 1\%$, we reduce preprocessing time by a factor of 30 and search space size by 240. We can only compare with TCH for approximated edge TTFs, as TCH computes the search spaces at query time and does not approximate any intermediate result. For $\varepsilon_e = 1\%$, we are 27 times faster than TCH (2.94 ms). But it pays off to approximate the search space TTFs, for $\varepsilon_e = \varepsilon_s = 0.1\%$, we are 56 times faster than TCH and even have smaller error (Table 6.18). Usually we would expect that the query time is independent of the table size, however, due to cache effects, we see an increase with increasing table size and a decrease with increasing ε's. Still the number of TTF evaluations is around 10 and thus 5 times large than the optimal (just 2).

By storing the minimal candidate (MINCANDIDATE, Table 6.3), we can reduce the evaluations to 3.7, which also reduces the query time. However, the precomputation is in $\Theta(|S| \cdot |T|)$, but it only becomes significant for size 10 000 (or larger). The time query is only about one third faster, as we still scan on average about 310 nodes in the forward/backward search spaces. For exact profile queries, there is no advantage to INTERSECT as we can afford ε_p-bound pruning at query time there.

Algorithm RELEVANTCANDIDATE (Table 6.4) makes scanning obsolete. It stores 1.2–3.4 candidates per source/target-pair, depending on used approximations. This is significantly smaller than the 20 meeting nodes we had before, reducing the time query by a factor of 2–4. But being in $\Theta(|S| \cdot |T|)$ becomes already noticeable for size 1 000. Again, the exact profile query does not benefit. Due to the knowledge of *all* relevant candidates, we only need to store 12% of all computed search space TTFs for size 100. This percentage increases naturally when the table size increases, or when we use worse bounds for pruning (larger ε_p), allowing a flexible trade-off between preprocessing time and space. Note that this algorithm can be used to make the INTERSECT algorithm more space-efficient by storing only the required TTFs and dropping c_{rel}.

We reduce the time query below 1 μs for any tested configuration with the OPT-

[2]We compare ourselves to TCH as it is currently the fastest exact speed-up technique.

Table 6.4: Performance of the RELEVANTCANDIDATE algorithm.

				preprocessing					search spaces				query		
size	ε_e	ε_s	ε_p	search	link	RAM	c_{rel}		[MiB]	TTF		point	time	eval	profile
	[%]	[%]	[%]	[s]	[s]	[MiB]	[MiB]	#		#	[%]	#	[μs]	#	[μs]
100	-	-	0.1	7.5	0.3	6 517	1	1.2	246	21	12	3 453	0.72	2.26	1 202
1 000	-	-	0.1	68.0	59.7	35 550	32	1.2	3 565	40	23	2 690	1.29	2.27	1 412
1 000	-	-	1.0	67.4	14.2	23 931	34	1.4	4 195	46	27	2 737	1.36	2.55	2 032
1 000	-	-	10.0	65.9	8.8	22 369	49	3.3	8 965	82	47	3 277	2.00	3.71	7 484
1 000	-	-	-	53.1	6.1	20 879	49	3.4	9 343	86	50	3 264	2.00	3.72	7 651
1 000	1.0	1.0	-	1.8	5.0	1 400	46	3.0	31	82	47	5.5	1.02	3.76	10.5
10 000	1.0	1.0	-	18.2	651.3	9 310	4 605	3.0	415	110	63	5.6	1.71	3.80	11.7

Table 6.5: Performance of the OPTCANDIDATE algorithm.

				preprocessing					search spaces				query	
size	ε_e	ε_s	ε_p	search	link	RAM	c_{opt}		[MiB]	TTF		point	time	profile
	[%]	[%]	[%]	[s]	[s]	[MiB]	[MiB]	#		#	[%]	#	[μs]	[μs]
100	-	-	0.1	7.5	2.1	6 494	1	1.5	241	21	12	3 456	0.49	1 168
500	-	-	0.1	33.8	53.7	13 101	10	1.5	1 608	33	19	2 946	0.73	1 391
1 000	-	-	0.1	68.0	213.8	21 443	39	1.5	3 489	39	22	2 704	0.81	1 332
1 000	-	-	-	53.1	1 032.2	20 908	39	1.5	3 489	39	22	2 704	0.84	1 339
1 000	1.0	-	-	1.5	19.4	1 666	37	1.4	72	39	23	49.5	0.56	21.6
1 000	-	1.0	-	64.9	7.1	5 318	39	1.5	17	41	23	6.0	0.51	3.5
1 000	0.1	0.1	-	4.7	11.5	1 822	39	1.5	42	39	22	25.9	0.54	9.8
1 000	1.0	1.0	-	1.8	6.3	1 385	38	1.5	15	41	24	4.9	0.48	3.0
1 000	10.0	10.0	-	0.7	7.6	963	62	3.1	19	68	39	2.0	0.49	5.7
10 000	1.0	1.0	-	18.2	788.3	7 741	3 775	1.5	226	63	36	5.0	0.90	3.5

CANDIDATE algorithm (Table 6.5),[3] as we always just need two TTF evaluations. Exact precomputation becomes very expensive due to the high number of TTF points, and pruning with ε_p-bounds has a big impact by almost a factor 4. However, in the heuristic scenario, precomputation is just around 25% slower than RELEVANTCANDIDATE (ε_p-pruning brings no speed-up), but provides more than 80% smaller search spaces and 3 times faster profile queries.

Naturally the best query times are achieved with the TABLE algorithm (Table 6.6). They are around a factor two smaller than OPTCANDIDATE, and up to 3 000 times faster than a TCH time query, and 4 000 000 times faster than a time-dependent Dijkstra. Note that we do not report profile query timings as they are a simple table look-up. The larger precomputation time compared to OPTCANDIDATE comes from the additional overhead to store the table. We cannot compute exact tables larger than size 500. But practical cases of size 1 000 can be computed with less than 2 GiB of RAM when we use

[3] $\#c_{opt} > \#c_{rel}$ is possible when candidates are optimal for several periods of time.

Table 6.6: Performance of the TABLE algorithm.

size	ε_e [%]	ε_s [%]	ε_p [%]	ε_t [%]	preprocessing search [s]	preprocessing link [s]	preprocessing RAM [MiB]	table [MiB]	table points #	query time [μs]
100	-	-	0.1	-	7.5	1.9	7 638	1 086	7 672	0.25
500	-	-	0.1	-	33.8	58.5	45 659	27 697	7 829	0.42
500	-	-	-	-	26.6	266.7	45 532	27 697	7 829	0.42
500	1.0	-	-	-	0.8	4.8	1 924	427	117.6	0.26
1 000	1.0	-	-	-	1.5	19.0	3 625	1 689	116.3	0.32
1 000	1.0	1.0	-	-	1.8	6.3	1 577	180	9.6	0.25
1 000	-	-	0.1	1.0	68.0	298.2	21 489	110	4.6	0.25
1 000	0.1	0.1	-	0.1	4.7	12.3	2 112	270	16.0	0.26
1 000	1.0	1.0	-	1.0	1.8	6.7	1 484	94	3.4	0.23
1 000	10.0	10.0	-	10.0	0.7	7.1	1 017	76	2.1	0.22
10 000	1.0	1.0	-	-	18.2	772.1	27 118	18 109	9.7	0.39
10 000	1.0	1.0	-	1.0	18.2	815.2	17 788	9 342	3.4	0.38

approximations (table TTFs are ε_t-approximations).

Compared to $|S| \cdot |T| = 500 \cdot 500$ exact TCH profile queries, taking 1023 s on 8 threads, our algorithm achieves a speed-up of 11. This speed-up increases for $\varepsilon_e = 1\%$ to 16 since there the duplicate work of the TCH queries has a larger share. And it increases further for table size 1 000, our speed-up is then 18, as our search increases linearly and only linking is quadratic in size. We were not able to compare ourselves to a plain single source Dijkstra profile query, as our implementation of it runs out of RAM (48 GiB) after around 30 minutes.

A visual comparison of all five algorithms is Figure 6.7. The vertical axis is relative to the maximum compared value in each group. The exact values can be found in Tables 6.2–6.6. We see that the decrease for time and profile query are almost independent of the size. However, the quadratic part for preprocessing time and space becomes very visible for size = 10 000.

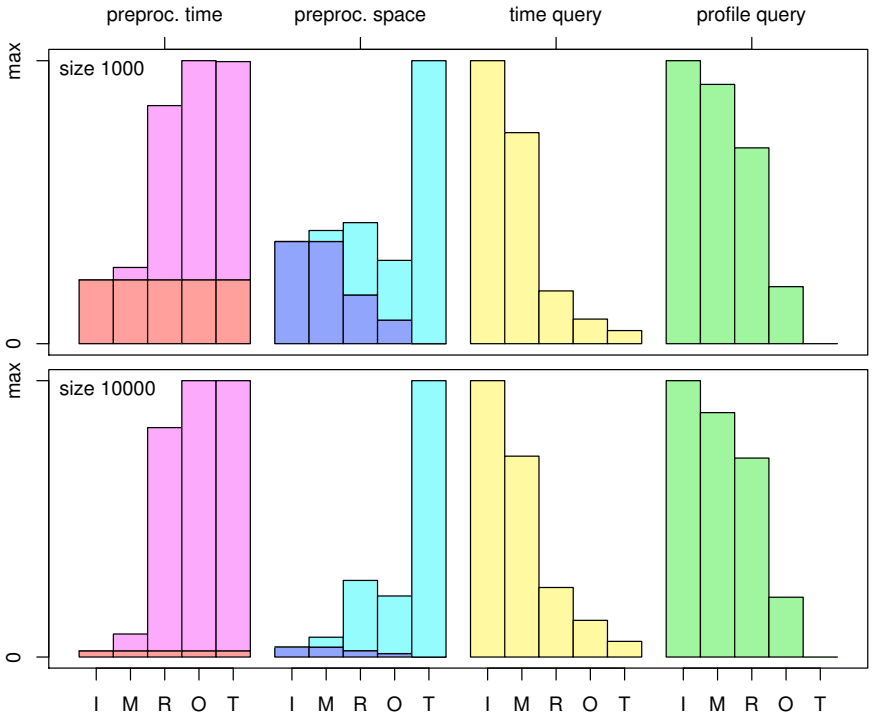


Figure 6.7: Comparison of the INTERSECT, MINCANDIATE, RELEVANTCANDIDATE, OPTCANDIDATE and TABLE algorithm with $\varepsilon_e = 1\%$, $\varepsilon_s = 1\%$, $\varepsilon_p = 0\%$ and $\varepsilon_t = 0\%$. Preprocessing time is split into search (lower) and link (upper) and space is split into TTFs (lower) and additional data (upper).

Robustness. To assess the robustness of our algorithms on other inputs, we performed additional experiments on TD-EUR and TD-GER-SUN. We restrict our selves to 20 000 time queries and 1 000 profile queries, otherwise the basic setup remains as for TD-GER. Our evaluation begins with TD-EUR, a larger graph with more time-dependency. There is one table of test results per algorithm, Tables 6.8–6.12. We achieve even better speed-ups over TCH than for TD-GER. The exact profile query of the INTERSECT algorithm has increased speed-up compared to TCH (382.12 ms), it is now 70 times faster. The heuristic profile query with $\varepsilon_e = 1\%$ is even 78 times faster than TCH (105.72 ms). The time queries are very time consuming for INTERSECT as there are 5 times more candidate nodes and 4 times more TTF evaluations necessary. However, we increase the speed-up of time queries for the OPTCANDIDATE algorithm, we are more than three orders of magnitude faster than TCH (1.89 ms). We were not able to compute an exact $500 \cdot 500$ TABLE as more than 48 GiB RAM are necessary. So we compare our table computation to TCH for size 100. $100 \cdot 100$ exact TCH profile queries, taking 478 s on 8 threads, are 9 times slower than our algorithm. For size 1 000 and $\varepsilon_e = 1\%$, we obtain a speed-up of 64 compared to $1\,000 \cdot 1\,000$ TCH profile queries on 8 threads, taking 13215 s. This is much larger than the speed-up of 16 for a comparable table on TD-GER.

Table 6.8: Performance of the INTERSECT algorithm on TD-EUR.

				preprocessing		search spaces			query					
size	ε_e	ε_s	ε_p	search	RAM	[MiB]	TTF	point	time	scan	meet	eval	succ	profile
	[%]	[%]	[%]	[s]	[MiB]		#	#	[μs]	#	#	#	#	[μs]
100	-	-	0.1	43.5	16 528	4 392	427	2 982	23.46	714	94.9	38.72	6.03	5 190
500	-	-	0.1	198.2	33 545	21 428	434	2 856	27.72	738	96.1	39.12	5.94	5 466
1 000	1.0	-	-	17.6	6 077	1 665	442	106.9	21.02	749	98.1	39.79	5.99	1 339.4
1 000	-	1.0	-	409.8	13 837	237	442	9.8	23.12	749	98.1	40.47	6.00	163.8
1 000	0.1	0.1	-	38.6	5 907	554	442	31.3	21.07	749	98.1	39.56	5.98	415.0
1 000	1.0	1.0	-	19.1	4 720	235	442	9.7	19.10	749	98.1	40.63	6.00	158.9
1 000	10.0	10.0	-	7.3	3 709	132	442	2.6	16.35	749	98.1	37.07	6.10	63.9
10 000	1.0	1.0	-	161.1	6 810	2 050	446	9.7	21.17	758	99.0	40.33	5.99	161.7

Table 6.9: Performance of the MINCANDIDATE algorithm on TD-EUR.

			preprocessing				search	query					
size	ε_e	ε_s	search	link	RAM	c_{min}	space	time	scan	meet	eval	succ	profile
	[%]	[%]	[s]	[s]	[MiB]	[MiB]	[MiB]	[μs]	#	#	#	#	[μs]
100	-	-	40.1	0.0	16 408	1	4 246	13.96	714	94.9	23.55	1.35	33 397
1 000	1.0	1.0	19.1	1.1	4 725	7	235	15.01	749	98.1	25.28	1.40	140.5
10 000	1.0	1.0	161.1	86.5	7 342	488	2 050	16.10	758	99.0	25.09	1.39	143.2

Table 6.10: Performance of the RELEVANTCANDIDATE algorithm on TD-EUR.

				preprocessing					search spaces				query		
size	ε_e	ε_s	ε_p	search	link	RAM	c_{rel}		[MiB]	TTF		point	time	eval	profile
	[%]	[%]	[%]	[s]	[s]	[MiB]	[MiB]	#		#	[%]	#	[μs]	#	[μs]
100	-	-	0.1	43.5	1.6	17 168	1	2.8	391	35	8	3 342	1.71	5.37	3 933
1 000	1.0	1.0	-	19.1	14.8	4 974	211	24.6	144	274	62	9.9	7.68	25.28	140.4
10 000	1.0	1.0	-	161.1	1 318.9	30 431	15 744	24.5	1 524	332	74	10.1	9.97	25.07	150.0

Table 6.11: Performance of the OPTCANDIDATE algorithm on TD-EUR.

				preprocessing					search spaces				query	
size	ε_e	ε_s	ε_p	search	link	RAM	c_{opt}		[MiB]	TTF		point	time	profile
	[%]	[%]	[%]	[s]	[s]	[MiB]	[MiB]	#		#	[%]	#	[μs]	[μs]
100	-	-	0.1	43.5	8.6	16 563	1	3.2	312	28	7	3 322	0.57	2 395
500	-	-	0.1	198.2	222.4	33 645	17	3.4	1 988	43	10	2 767	0.73	2 452
1 000	1.0	-	-	17.6	189.9	6 213	83	4.4	181	53	12	95.6	0.73	95.1
1 000	-	1.0	-	409.8	35.2	14 290	105	5.9	32	62	14	9.5	0.79	15.4
1 000	0.1	0.1	-	38.6	68.5	5 982	82	4.3	63	53	12	29.7	0.66	28.8
1 000	1.0	1.0	-	19.1	34.0	4 808	104	5.8	32	63	14	9.4	0.60	14.8
1 000	10.0	10.0	-	7.3	21.7	3 864	80	4.2	23	83	19	2.3	0.55	10.6
10 000	1.0	1.0	-	161.1	2 713.5	16 233	7 696	5.7	397	89	20	9.6	1.03	15.7

Table 6.12: Performance of the TABLE algorithm on TD-EUR.

					preprocessing			table		query
size	ε_e	ε_s	ε_p	ε_t	search	link	RAM	[MiB]	points	time
	[%]	[%]	[%]	[%]	[s]	[s]	[MiB]		#	[μs]
100	-	-	0.1	-	43.5	7.1	16 636	832	5 877	0.24
500	1.0	-	-	-	8.7	45.8	6 122	741	206.3	0.28
1 000	1.0	-	-	-	17.6	184.5	9 884	2 980	207.4	0.34
1 000	1.0	1.0	-	-	19.1	32.6	5 148	391	24.5	0.25
1 000	0.1	0.1	-	0.1	38.6	67.5	6 201	393	24.6	0.26
1 000	1.0	1.0	-	1.0	19.1	33.4	4 884	151	7.6	0.23
1 000	10.0	10.0	-	10.0	7.3	19.9	3 892	75	2.0	0.17
10 000	1.0	1.0	-	1.0	161.1	2 689.5	23 696	11 260	7.5	0.42

Our experiments on TD-GER-SUN show the automatic adaptability of our algorithms to easier instances. TD-GER-SUN has particularly simple TTFs, as the travel times only slightly change over time, and even the exact graph requires little space (Table 6.1). There is one table of test results per algorithm, Tables 6.13–6.17. We notice that profile queries are one order of magnitude faster than in the midweek scenario. Still, they benefit from ε_p-bound pruning, but not as much as before. Also, computing whole tables is much faster and requires much less RAM. We need only 7 GiB RAM to compute an exact $500 \cdot 500$ table, instead of 46 GiB for TD-GER. The reason is that the TTFs in the table have more than 7 times fewer points. Even the computation of an exact $1\,000 \cdot 1\,000$ table is possible with less than 22 GiB RAM.

Table 6.13: Performance of the INTERSECT algorithm on TD-GER-SUN.

				preprocessing		search spaces			query					
size	ε_e	ε_s	ε_p	search	RAM	[MiB]	TTF	point	time	scan	meet	eval	succ	profile
	[%]	[%]	[%]	[s]	[MiB]		#	#	[μs]	#	#	#	#	[μs]
100	-	-	0.1	0.6	1 744	201	117	459	2.50	214	12.8	6.65	3.21	133
500	-	-	0.1	2.5	2 518	979	117	449	4.08	214	12.7	6.79	3.28	151
1 000	-	-	0.1	4.7	3 489	1 930	117	443	4.61	214	12.8	6.68	3.22	140
1 000	-	-	-	3.0	3 295	1 750	117	443	4.71	214	12.8	6.68	3.22	210
1 000	1.0	-	-	0.3	815	87	117	16.4	3.55	214	12.8	6.73	3.22	10.9
1 000	-	1.0	-	4.3	1 605	34	117	2.8	3.56	214	12.8	6.83	3.23	5.3
1 000	0.1	0.1	-	0.8	902	65	117	10.8	3.58	214	12.8	6.69	3.22	7.6
1 000	1.0	1.0	-	0.4	764	33	117	2.5	3.31	214	12.8	6.87	3.23	5.2
1 000	10.0	10.0	-	0.3	649	30	117	1.8	3.98	214	12.8	11.15	3.25	14.0
10 000	1.0	1.0	-	3.4	1 079	281	119	2.5	4.59	216	12.8	6.86	3.23	6.2

Table 6.14: Performance of the MINCANDIDATE algorithm on TD-GER-SUN.

			preprocessing				search	query					
size	ε_e	ε_s	search	link	RAM	c_{min}	space	time	scan	meet	eval	succ	profile
	[%]	[%]	[s]	[s]	[MiB]	[MiB]	[MiB]	[μs]	#	#	#	#	[μs]
100	-	-	0.3	0.0	1 722	1	182	1.70	214	12.8	2.16	1.01	192
1 000	-	-	3.0	0.2	3 297	7	1 750	3.05	214	12.8	2.17	1.01	207
1 000	1.0	1.0	0.4	0.2	773	7	33	2.73	214	12.8	2.33	1.03	4.3
10 000	1.0	1.0	3.4	26.3	1 648	484	281	3.26	216	12.8	2.33	1.03	4.9

Table 6.15: Performance of the RELEVANTCANDIDATE algorithm on TD-GER-SUN.

				preprocessing					search spaces				query		
size	ε_e	ε_s	ε_p	search	link	RAM	c_{rel}		[MiB]	TTF		point	time	eval	profile
	[%]	[%]	[%]	[s]	[s]	[MiB]	[MiB]	#		#	[%]	#	[μs]	#	[μs]
100	-	-	0.1	0.6	0.0	1 746	1	1.0	36	19	16	563	0.34	2.06	123
1 000	-	-	0.1	4.7	7.1	5 150	31	1.1	590	37	32	471	0.92	2.07	138
1 000	-	-	1.0	4.5	3.8	3 909	32	1.1	650	40	34	483	0.95	2.15	187
1 000	-	-	10.0	4.7	3.5	3 793	32	1.2	784	44	38	525	0.94	2.17	220
1 000	-	-	-	3.0	3.0	3 365	32	1.2	785	44	38	525	0.88	2.17	220
1 000	1.0	1.0	-	0.4	2.8	852	34	1.4	13	47	40	2.5	0.54	2.33	2.0
10 000	1.0	1.0	-	3.4	312.1	5 382	2 503	1.4	157	67	57	2.6	1.02	2.33	2.5

Table 6.16: Performance of the OPTCANDIDATE algorithm on TD-GER-SUN.

				preprocessing					search spaces				query	
size	ε_e	ε_s	ε_p	search	link	RAM	c_{opt}		[MiB]	TTF		point	time	profile
	[%]	[%]	[%]	[s]	[s]	[MiB]	[MiB]	#		#	[%]	#	[μs]	[μs]
100	-	-	0.1	0.6	0.2	1 744	1	1.1	36	19	16	566	0.25	119
500	-	-	0.1	2.5	5.4	2 550	9	1.1	256	30	26	502	0.54	136
1 000	-	-	0.1	4.7	22.0	3 572	33	1.1	575	36	31	470	0.65	132
1 000	-	-	-	3.0	30.5	3 357	33	1.1	575	36	31	470	0.63	131
1 000	1.0	-	-	0.3	3.8	886	32	1.1	27	37	31	16.4	0.46	5.4
1 000	-	1.0	-	4.3	3.3	1 638	33	1.1	11	37	32	2.6	0.38	1.4
1 000	0.1	0.1	-	0.8	3.4	971	32	1.1	19	37	31	9.9	0.45	3.0
1 000	1.0	1.0	-	0.4	3.1	834	33	1.1	11	38	32	2.4	0.38	1.4
1 000	10.0	10.0	-	0.3	4.9	737	52	2.4	15	57	48	1.9	0.43	3.6
10 000	1.0	1.0	-	3.4	333.2	4 852	2 411	1.1	127	56	47	2.5	0.80	1.8

Table 6.17: Performance of the TABLE algorithm on TD-GER-SUN.

					preprocessing			table		query
size	ε_e	ε_s	ε_p	ε_t	search	link	RAM	[MiB]	points	time
	[%]	[%]	[%]	[%]	[s]	[s]	[MiB]		#	[μs]
100	-	-	0.1	-	0.6	0.2	1 909	154	1 083	0.14
500	-	-	0.1	-	2.5	6.6	7 049	3 829	1 079	0.30
1 000	-	-	0.1	-	4.7	27.5	21 193	14 986	1 056	0.38
500	-	-	-	-	1.6	8.1	6 944	3 829	1 079	0.30
500	1.0	-	-	-	0.2	0.9	946	137	35.4	0.20
1 000	1.0	-	-	-	0.3	4.0	1 502	543	35.2	0.25
1 000	1.0	1.0	-	-	0.4	3.1	948	108	4.5	0.19
1 000	-	-	0.1	1.0	4.7	35.3	3 650	77	2.2	0.16
1 000	0.1	0.1	-	0.1	0.8	3.9	1 101	121	5.4	0.21
1 000	1.0	1.0	-	1.0	0.4	3.3	917	76	2.1	0.17
1 000	10.0	10.0	-	10.0	0.3	4.4	801	76	2.0	0.17
10 000	1.0	1.0	-	1.0	3.4	337.6	11 133	5 658	2.1	0.33

Error Analysis. We analyze the observed errors and theoretical error bounds[4] for size 1 000 in Table 6.18. Note that these error bounds are independent of the used algorithm. The maximum slope[5] for TD-GER is $\alpha = 0.433872$. The average observed error is always below the used ε's, however, we still see the stacking effect. Our bound is about a factor of two larger than the maximum observed error for the edge approximations (ε_e). This is because our bound assumes that any error stacks during the linking of the TTFs, but in practice, TTFs do not often significantly change. The same explanation holds for the search space approximations (ε_s), although our bound is better since we only need to link two TTFs. When we combine edge and search space approximations, the errors roughly add up, as approximating an already approximated TTF introduces new errors. Approximating the TTFs in the table (ε_t) gives the straight ε_t-approximation unless it is based on already approximated TTFs. Our theoretical bounds are pretty tight for the tested TCH instance, just around a factor of two larger than the max. observed errors.

For TD-EUR and TD-GER-SUN we observe similar average and maximum errors as for TD-GER. However, there are significant differences for the theoretical bounds. The maximum slope α of the TTFs in the search spaces on TD-EUR is at least[6] 5.82697, resulting in very large upper bounds. In practice, we do not observe these large errors, as only small parts of the TTFs have such big slopes. The big slopes may be an artifact of the artificially created TTFs on this graph. However, in practice, such big slopes can also occur, when e. g. ferries with time tables would be included. A more detailed error model, not only based on the maximum slope α, is necessary to provide better theoretical bounds in such a scenario. In contrast, we can provide very tight error bounds for TD-GER-SUN, as the maximum slope is just $\alpha = 0.137686$. This is due to the much "flatter" TTFs on Sunday, as very little time-dependency exists.

Table 6.18: Observed errors from our queries together with the theoretical error bounds.

	ε_e [%]	1.0	-	0.1	1.0	10	-	0.1	1.0	10
	ε_s [%]	-	1.0	0.1	1.0	10	-	0.1	1.0	10
graph	ε_t [%]	-	-	-	-	-	1.0	0.1	1.0	10
TD-GER	avg. error [%]	0.08	0.12	0.014	0.18	2.1	0.17	0.023	0.30	3.1
	max. error [%]	0.89	0.98	0.169	1.75	16.9	1.00	0.266	2.66	24.9
	theo. bound [%]	2.07	1.44	0.350	3.55	41.0	1.00	0.450	4.58	55.1
TD-EUR	avg. error [%]	0.10	0.22	0.025	0.24	2.7	0.30	0.038	0.38	4.8
	max. error [%]	0.87	1.31	0.155	1.77	15.0	1.00	0.228	2.75	22.4
	theo. bound[6] [%]	52.55	6.89	5.412	60.85	3 399.4	1.00	5.517	62.46	3 749.4
TD-GER-SUN	avg. error [%]	0.05	0.13	0.014	0.15	1.8	0.17	0.023	0.26	2.8
	max. error [%]	0.78	0.96	0.149	1.55	16.2	1.00	0.237	2.33	24.1
	theo. bound [%]	1.30	1.14	0.243	2.45	26.5	1.00	0.344	3.48	39.2

[4]Note that $\varepsilon_p > 0$ used for pruning does not cause errors.

[5]It is *only* required for the theoretical error bounds, and not used in our algorithms.

[6] The slope is only the maximum over all search spaces computed during the experiments. Computing the exact slope would take more than a week.

6.3 Ride Sharing

In the ride sharing scenario, we want to match a *driver* (with a car), and a *passenger* (without a car) so that they can share some part of their journey, and both have an economical advantage of this cooperation. There exist a number of web sites that offer ride sharing matching services to their customers. Unfortunately, as far as we know, all of them suffer from limitations in their method of matching. Only a very small and limited subset of all the possible locations is actually modeled. Furthermore, a match is only based on the proximity of the starting locations of driver and passenger, and on their destinations. Sometimes, some intermediate stops are given beforehand. The economic advantage is largely neglected. Consider the following example to visualize this. Anne and Bob both live in Germany. Anne, the driver, is from Karlsruhe and wants to go to Berlin. Bob on the other hand lives in Frankfurt and would like to travel to Leipzig. Taking the fastest route in our example, Anne drives from Karlsruhe via Nürnberg to Berlin and is never getting close enough to team up with Bob. However, there is a path from Karlsruhe to Berlin via Frankfurt, which also passes by the city of Leipzig and is only about one percent longer than the shortest path. We propose to match a driver and passenger, so that the detour of the driver to serve the passenger is minimized. This detour is usually directly linked to the economic advantage. Such a matching algorithm only becomes feasible by employing fast batched shortest paths algorithms. In principle, we compute the detours for all offering drivers, and pick the minimum. Additionally, we can prune some computations by limiting the maximum allowed detour.

6.3.1 Matching Definition

For many services an offer only fits a request iff origin and destination locations are identical. We call such a situation a *perfect fit*. Origin and destination can only be chosen from a limited set of points, for example all larger cities and points-of-interest such as airports. Some services offer an additional radial search around origin and destination and fixed way-points along the route. Usually, only the driver is able to define way-points along her route. The existence of these additions shows the demand for better route matching techniques that allow a small detour and intermediate stops. We call that kind of matching a *reasonable fit*. However, previous approaches obviously used only features of the database systems they had available to compute matches. And we showed in the previous section that the previous approaches are not flexible and miss possibly interesting matches.

We present an algorithmic solution to the situation at hand that leads to better results independent of the user's level of cooperation or available database systems. For that, we lift the restriction of a limited set of origin and destination points. Unfortunately, the probability of perfect fits is close to zero in this setting. But since we want to compute reasonable fits, our approach considers intermediate stops where driver and passenger

might meet and depart later on. More precisely, we adjust the drivers route to pick up the passenger by allowing an acceptable detour (Definition 6.7).

Definition 6.7 *Let G be a static graph (Section 2.2.1). We say that an offer $o = (s,t)$ and a request $g = (s',t')$ form a* reasonable fit *iff there exists a path $P = \langle s,\ldots,s',\ldots,t',\ldots,t\rangle$ in G with $c(P) \leq \mu(s,t) + \varepsilon \cdot \mu(s',t')$.*

The ε in Definition 6.7 depicts the maximal detour that is reasonable. Applying the ε to the passengers path gives the driver an incentive to pick up the passenger. A natural choice is $\varepsilon \leq 0.5$. This stems from a simple pricing scheme we know from algorithmic game theory. The so-called *fair sharing rule* [114] simply states that players who share a ride split costs evenly for the duration of the ride. Additionally, we say that drivers get compensated for their detours directly by passengers using the savings from the shared ride. Implicitly, we give the driver an incentive to actually pick the passengers up at their start s' and to drop them off at their destination t'. Formally, we have that a match is economically worthwhile iff there exists an ε for which

$$\mu(s,s') + \mu(s',t') + \mu(s',t') + \mu(t',t) - \mu(s,t) \leq \varepsilon \cdot \mu(s',t') \quad .$$

It is easy to see that any reasonable passenger will not pay more for the drivers detour than the gain for the shared ride which is at most $\frac{1}{2} \cdot \mu(s',t')$. Therefore, we conclude $\varepsilon \leq 0.5$. Figure 6.19 details the distances of the argument:

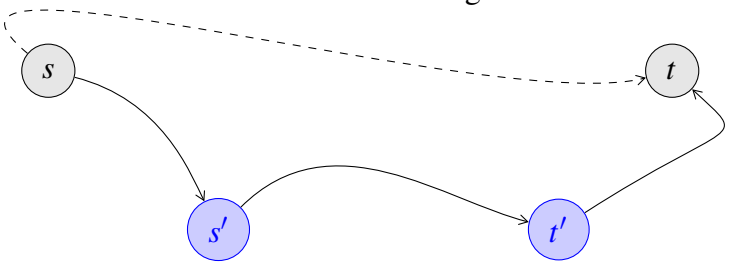


Figure 6.19: Request (s',t') and matching offer (s,t) with detour. The solid lines symbolize the distances that are driven, while the dashed one stands for the shortest path of the driver that is actually not driven at all in a matched ride.

6.3.2 Matching Algorithm

This section covers the algorithm to find all reasonable fits to an offer. We even solve the more general problem of computing all detours.

For a dataset of k offers $o_i = (s_i,t_i)$, i=1..k, and a single request $g = (s',t')$, we need to compute the $2k+1$ shortest path distances $\mu(s',t')$, $\mu(s_i,s')$ and $\mu(t',t_i)$. The detour for offer o_i is then $\mu(s_i,s') + \mu(s',t') + \mu(t',t_i) - \mu(s_i,t_i)$. A naïve algorithm would do a backward one-to-all search from s' using Dijkstra's algorithm and a forward

one-to-all search from t' to compute $\mu(s_i, s')$ and $\mu(t', t_i)$. Another search returns the distance $\mu(s', t')$. We cannot prune the Dijkstra search early, as the best offer need not depart/arrive near the source/target of the request, so that each search takes several seconds on large road networks. In Section 6.3.3 we show that the running time of our algorithm is faster by several orders of magnitude.

To compute the distances, we use the technique described in Section 6.1.1. We store the forward search space from each s_i, i=1..k, in *forward buckets* (6.4) to compute $\mu(s_i, s')$ using (6.7). And we store the backward search space from each t_i, i=1..k, in *backward buckets* (6.5) to compute $\mu(t', t_i)$ using (6.6).

However, in the buckets, we store references to the offer o_i instead of the source node s_i or target node t_i. More precisely, our *forward bucket* for a node u is

$$\overrightarrow{\beta}(u) := \left\{ (i, \overrightarrow{\delta}_i(u)) \mid (u, \overrightarrow{\delta}_i(u)) \in \overrightarrow{\sigma}(s_i) \right\} \quad . \tag{6.8}$$

Here, $\overrightarrow{\delta}_i(u)$ denotes the distance from s_i to u computed by the forward search from s_i.

To compute all $\mu(s_i, s')$ for the request, we compute $\overleftarrow{\sigma}(s')$, then scan the bucket of each node in $\overleftarrow{\sigma}(s')$ and compute all $\mu(s_i, s')$ simultaneously as described in Section 6.1.1.

Symmetrically, we compute and store *backward buckets*

$$\overleftarrow{\beta}(u) := \left\{ (i, \overleftarrow{\delta}_i(u)) \mid (u, \overleftarrow{\delta}_i(u)) \in \overleftarrow{\sigma}(t_i) \right\} \tag{6.9}$$

to accelerate the computation of all $\mu(t', t_i)$. The single distance $\mu(s', t')$ is computed separately by computing the search spaces from s' and t' in the opposite directions. Backward and forward buckets are stored in main memory and accessed as our main data structure and running queries on that data structure is easy.

Adding and Removing Offers. To add or remove an offer $o = (s, t)$, we only need to update the forward and backward buckets. To add the offer, we first compute $\overrightarrow{\sigma}(s)$ and $\overleftarrow{\sigma}(t)$. We then add these entries to their corresponding forward/backward buckets. To remove the offer, we need to remove its entries from the forward/backward buckets.

We make no decision on the order in which to store the entries of a bucket. This makes adding an offer very fast, but removing it requires scanning the buckets. Scanning all buckets is prohibitive as there are too many entries. Instead, it is faster to compute $\overrightarrow{\sigma}(s)$ and $\overleftarrow{\sigma}(t)$ again to obtain the set of meeting nodes whose buckets contain an entry about this offer. We then just need to scan those buckets and remove the entries. Also, we can omit removing offers by just having a separate bucket for each day. We mark obsolete offers so that they will be ignored for any follow-up requests, and delete the whole bucket once the day of the bucket is in the past.

Constraints. In reality, offers and requests have constraints. For example, they specify a departure time window or they have restrictions on smoking, gender, etc. In this case,

we need to extend the definition of a reasonable fit to meet these constraints by introducing additional indicator variables. As we already compute the detours of all offers, we can just filter the ones that violate the constraints of the request. Furthermore, our algorithm can potentially take advantage of these constraints by storing offers in different buckets depending on their constraints. This way, we reduce the number of bucket entries that are scanned during a request, reducing the time to match a request as the bucket scans take the majority of the time.

Algorithmic Optimizations. We reduce the request matching time by pruning bucket scans. We can omit scanning buckets when we limit the maximum detour to ε times the cost of the passengers shortest route, as stated in Definition 6.7. We exploit the fact that we need to obtain $\overrightarrow{\sigma}(s')$ and $\overleftarrow{\sigma}(t')$ for the computation of $\mu(s',t')$. We compute the distance $\mu(s',t')$ before the bucket scanning, and additionally keep $\overrightarrow{\sigma}(s')$ and $\overleftarrow{\sigma}(t')$ that we obtained during this search. Then we can apply Lemma 6.8. It can be applied if a node u appears in both backward search $\overleftarrow{\sigma}(s')$, $\overleftarrow{\sigma}(t')$, or in both forward search spaces $\overrightarrow{\sigma}(s')$, $\overrightarrow{\sigma}(t')$. In this case, we can compute a lower bound on the detour of all offers in the bucket of u by using the triangle inequality, see Figure 6.20. If this lower bound is larger than the detour allowed by ε, we prune the bucket.

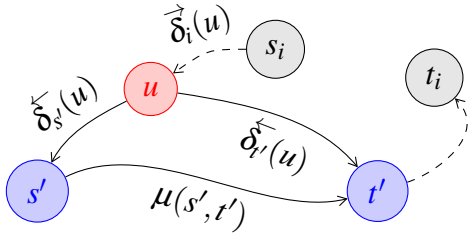


Figure 6.20: The difference $\overleftarrow{\delta}_{s'}(u) + \mu(s',t') - \overleftarrow{\delta}_{t'}(u)$ is a lower bound on a detour via u.

Lemma 6.8 *Let $(u, \overleftarrow{\delta}_{s'}(u)) \in \overleftarrow{\sigma}(s')$ and $(u, \overleftarrow{\delta}_{t'}(u)) \in \overleftarrow{\sigma}(t')$. We will not miss a reasonable fit when we omit scanning forward bucket $\overrightarrow{\beta}(u)$ only if $\overleftarrow{\delta}_{s'}(u) + \mu(s',t') > \overleftarrow{\delta}_{t'}(u) + \varepsilon \cdot \mu(s',t')$.*

Let $(u, \overrightarrow{\delta}_{t'}(u)) \in \overrightarrow{\sigma}(t')$ and $(u, \overrightarrow{\delta}_{s'}(u)) \in \overrightarrow{\sigma}(s')$. We will not miss a reasonable fit when we omit scanning backward bucket $\overleftarrow{\beta}(u)$ only if $\overrightarrow{\delta}_{t'}(u) + \mu(s',t') > \overrightarrow{\delta}_{s'}(u) + \varepsilon \cdot \mu(s',t')$.

Proof. Let $(i, \overrightarrow{\delta}_i(u)) \in \overrightarrow{\beta}(u)$ be a pruned offer. If the path from s_i to s' via node u is not a shortest path, another meeting node will have s_i in its bucket, see (6.8). Therefore, WLOG we assume that $\overrightarrow{\delta}_i(u) + \overleftarrow{\delta}_{s'}(u) = \mu(s_i, s')$. Let $P = \langle s_i, \ldots, s', \ldots, t', \ldots, t_i \rangle$ be a path, then

$$
\begin{array}{rcl}
c(P) & \geq & \mu(s_i,s') + \mu(s',t') + \mu(t',t_i) \\
 & = & \overrightarrow{\delta_i}(u) + \overleftarrow{\delta_{s'}}(u) + \mu(s',t') + \mu(t',t_i) \\
 & \overset{\text{Lemma 6.8}}{>} & \overrightarrow{\delta_i}(u) + \overleftarrow{\delta_{t'}}(u) + \varepsilon \cdot \mu(s',t') + \mu(t',t_i) \\
 & \overset{\overrightarrow{\delta_i}(u) \geq \mu(s_i,u),\ \overleftarrow{\delta_{t'}}(u) \geq \mu(u,t')}{\geq} & (\mu(s_i,u) + \mu(u,t') + \mu(t',t_i)) + \varepsilon \cdot \mu(s',t') \\
 & \overset{\triangle\text{-inequality}}{\geq} & \mu(s_i,t_i) + \varepsilon \cdot \mu(s',t')
\end{array}
$$

Therefore, P is not a reasonable fit. The proof is completely symmetric for omitting the scan of $\overleftarrow{\beta}(u)$. □

6.3.3 Experiments

Environment. Experiments have been done on one core of a single AMD Opteron Processor 270 clocked at 2.0 GHz with 8 GiB main memory and 2×1 MiB L2 cache, running SuSE Linux 11.1 (kernel 2.6.27). The program was compiled by the GNU C++ compiler 4.3.2 using optimization level 3.

Instances. We use a graph of Germany derived from the publicly available data of OpenStreetMap, consisting of 6 344 491 nodes and 13 513 085 directed edges. The edge weights are travel times computed for the OpenStreetMap car speed profile[7]. We use OpenStreetMap as source of our graph, as there are no licensing issues with using the data for non-academic purposes, for example to setup a website using our algorithm. Contraction hierarchies (aggressive approach [59]) are used as bi-directed, non goal-directed speed-up technique. To test our algorithm, we obtained a dataset of real-world ride sharing offers from Germany available on the web. We matched the data against a list of cities, islands, airports and the like, and ended up with about 450 unique places. We tested the data and checked that the lengths of the journeys are exponentially distributed. This validates assumptions from the field of transportation science. We assumed that requests would follow the same distribution and chose our offers from that dataset as well.

To extend the dataset to our approach of arbitrary origin and destination locations, we applied perturbation to the node locations of the dataset. For each source node we unpacked the node's forward search space in the contraction hierarchy up to a distance of 3 000 seconds of travel time. From that unpacked search space we randomly selected a new starting point. Likewise we unpacked the backward search space of each destination node up to the distance and picked a new destination node. We observed that perturbation preserved the distribution of the original dataset.

Figure 6.21 compares the original node locations on the left to the result of the node perturbation in the middle. The right side shows a population density plot of Germany[8] to support the validity of the perturbation.

[7] See: http://wiki.openstreetmap.org/wiki/OpenRouteService

[8] Picture is an extract of an image available at episcangis.hygiene.uni-wuerzburg.de

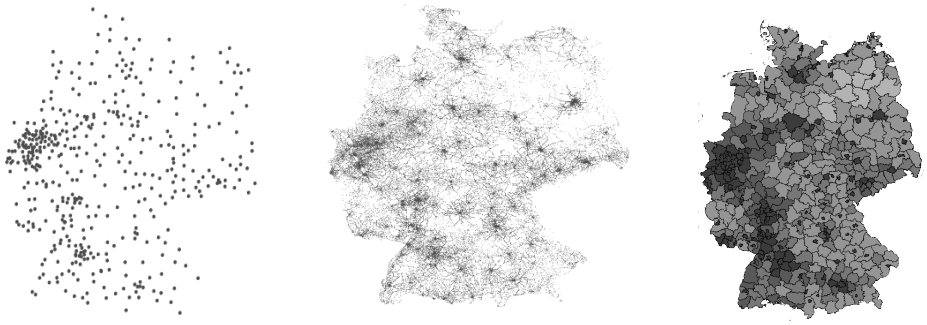


Figure 6.21: original node locations (left), perturbed node locations (middle), population density (right)

We evaluated the performance of our algorithm for different numbers of offers where source and target are picked at random or from our un-/perturbed real-world dataset, see Table 6.22. The size required for the bucket entries is linear with the number of offers, as a forward/backward search space has at most a few hundred nodes. The time to add an offer $o = (s,t)$ is independent of the number of offers, the main time is spent computing $\overrightarrow{\delta}(s)$ and $\overleftarrow{\delta}(t)$. However, removing an offer requires scanning the buckets, and therefore the more offers are in the database the more expensive it is. For our real-world offers, we have just 450 different source/target nodes, so that the bucket entries are clustered in only a few buckets, this still holds when we perturb the data. Of course, the bucket entries are more evenly distributed for completely random offers, the buckets are therefore smaller and removing an offer takes less time. We report the time for matching a request for different values of ε. Even with no further optimization ($\varepsilon = \infty$), we can handle large datasets with 100 000 offers within 45 ms. In comparison, the fastest speedup technique

Table 6.22: Performance of our algorithm for different types of offers/requests, numbers of offers and max. detours ε.

	#offers	bucket	add	remove	match request [ms]								
		size	offer	offer	$\varepsilon =$								
type		[MiB]	[ms]	[ms]	0.0	0.05	0.1	0.2	0.3	0.4	0.5	1	∞
perturbed	1 000	3	0.27	0.00	0.6	0.6	0.6	0.7	0.7	0.7	0.7	0.8	0.9
perturbed	10 000	28	0.24	0.29	0.9	1.0	1.1	1.3	1.5	1.6	1.8	2.7	4.1
perturbed	100 000	279	0.24	0.30	4.4	5.2	6.1	8.1	10.2	12.1	14.0	25.1	43.4
unperturbed	1 000	3	0.26	0.27	0.7	0.7	0.7	0.7	0.8	0.8	0.8	0.9	1.0
unperturbed	10 000	32	0.26	0.32	1.1	1.2	1.3	1.6	1.7	1.9	2.1	2.8	4.3
unperturbed	100 000	318	0.27	6.26	5.6	6.7	7.9	10.4	12.4	14.5	16.1	26.3	44.6
random	1 000	3	0.24	0.25	0.7	0.7	0.7	0.7	0.7	0.7	0.8	0.9	1.0
random	10 000	31	0.25	0.30	1.1	1.2	1.3	1.5	1.7	1.9	2.1	3.5	4.3
random	100 000	306	0.26	0.32	6.0	6.7	7.8	10.1	12.6	15.4	18.5	34.9	45.1

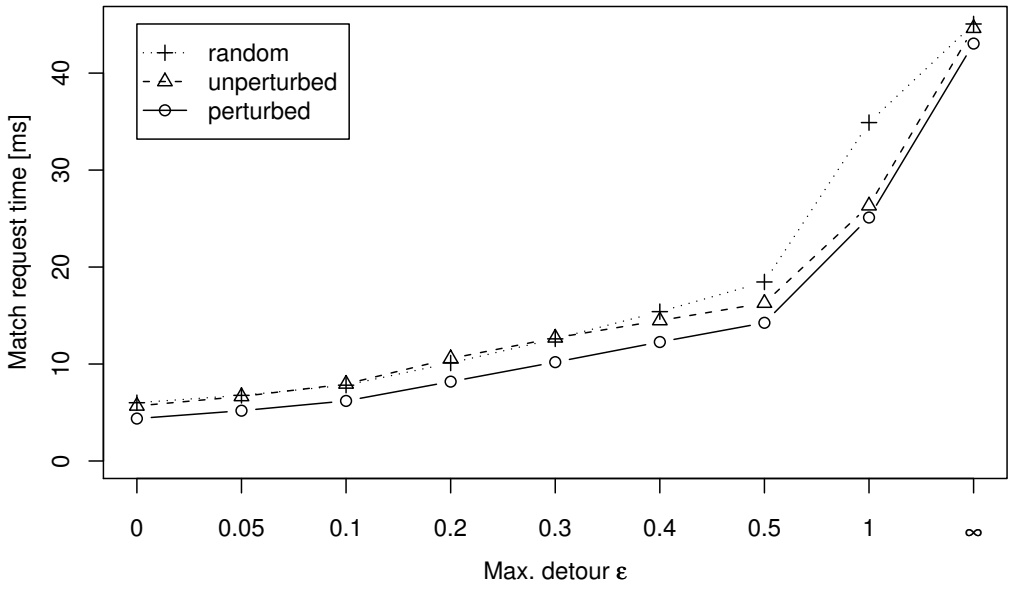


Figure 6.23: Match request performance for 100 000 offers.

today, Transit Node Routing (TNR) [22, 14] requires 1,9 μs[9] for each of the $2n+1$ queries and would take about 380 ms for the largest dataset whereas our algorithm is 8.4 times faster. For a realistic $\varepsilon = 0.5$, we get a further speed-up of about 3. Figure 6.23 visualizes the performance for different ε. It mainly depends on ε and our algorithm is fairly robust against the different ways to pick source and target nodes.

Our method is also faster than TNR when we look at preprocessing. Although TNR does not need to add and store offers, our algorithm based on CH is still faster. The preprocessing of CH is one order of magnitude faster and has no space overhead, whereas TNR would require more than 1 GiB on our graph of Germany. This is more than enough time to insert even 100 000 offers and more than enough space to store the bucket entries, as Table 6.22 indicates.

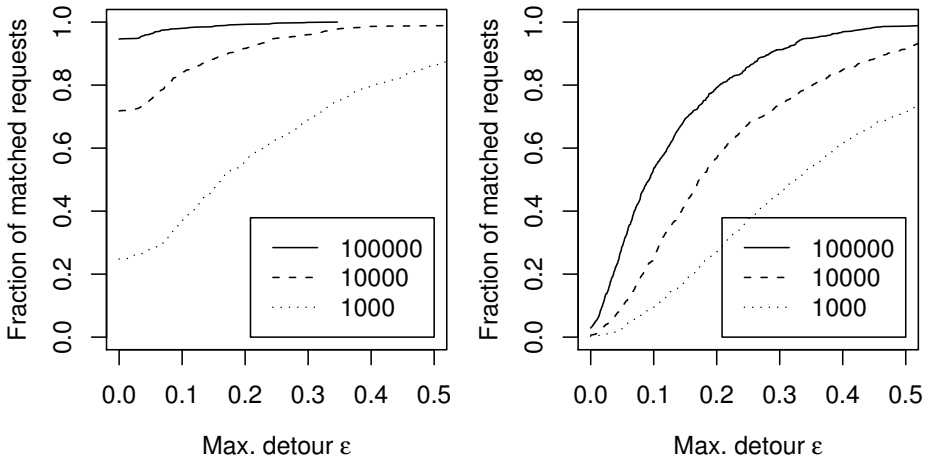
We varied the allowed detour and investigated what influence it has on the number of matches that can be made. A random but fixed sample of 1 000 requests was matched against databases of various sizes. Table 6.24 and Figure 6.25 report on these experiments. The unperturbed data presents the currently used algorithms, where you are able to do city-to-city queries. For a realistic database size of 10 000 entries[10] and maximum allowed detour of $\varepsilon = 0.1$ we improve the matching rate to 0.84. This is a lot more than the 0.718 matching rate without detours. As expected, the matching rate increases with the number of offers.

[9]This query time is on the European road network, but since the number of access nodes should be the same on Germany, we can expect a similar query time there.

[10]The database size of 10 000 entries is a realistic case and closely resembles the current daily amount of matches made by a known German ride sharing service provider, see: http://www.ea-media.net/geschaftsfelder/europealive/geschaftsfelder.html

Table 6.24: Request matching rate for different values of maximum allowed detour.

	UNPERTURBED					PERTURBED				
	$\varepsilon =$					$\varepsilon =$				
#offers	0	0.05	0.1	0.2	0.5	0	0.05	0.1	0.2	0.5
1000	0.248	0.281	0.370	0.558	0.865	0.003	0.028	0.096	0.271	0.715
10000	0.718	0.755	0.840	0.917	0.989	0.006	0.093	0.248	0.569	0.914
100000	0.946	0.963	0.981	0.993	1.000	0.029	0.289	0.537	0.793	0.988



(a) Unperturbed (b) Perturbed

Figure 6.25: Fraction of rides matched for a given detour.

Table 6.26: Detour sizes relative to the passengers route length for the best answer achieved using radial search and using our algorithm.

#offers	radial search detour	smallest detour
1 000	0.806	0.392
10 000	0.467	0.227
100 000	0.276	0.128

The more realistic scenario with the perturbed data, where offers and requests are not only city-to-city, but point-to-point, becomes only practically possible with our new algorithm. The probability to find a perfect match in this scenario is close to zero, it is necessary to allow at least a small detour to find some matches. The ε required to find a match becomes larger, as we now also include intra-city detours and not only inter-city detours. Still, with $\varepsilon = 0.2$ we achieve a matching rate of 0.569, and for the maximum reasonable detour of $\varepsilon = 0.5$, we match 0.914 of all requests, that is 20% more than the 0.718 possible with a city-to-city perfect matching algorithm (unperturbed, $\varepsilon = 0$).

We also tested the quality of our algorithm against radial search. In the radial search setting, each request is matched against the offer with the smallest sum of Euclidean distances w. r. t. the origin and destination location of the request. This mimics radial search functions (with user supplied radii) offered in some current ride sharing systems. Table 6.26 reports the results. The average detour of all matches is less than half the detour that is experienced with radial search, which shows the performance of our approach. On the other hand, these numbers show the inferiority of radial search.

6.4 Closest Point-of-Interest Location

Large point-of-interest (POI) databases are a crucial part of navigation systems and extend the usability beyond path directions. With them, the user is able to get along in unknown areas by listing the closest gas stations, garages, or even Italian restaurants. Fast computation of them is important for an interactive system. Dijkstra's algorithm can only efficiently compute POI that are very close to a node. Computing POI that are farther away, or close POI along a shortest path requires much more time. We will use techniques inspired by the ideas of Section 6.1 to accelerate both types of POI computations. We compute from each POI x the search space and store an entry in a bucket for each reached node u containing the computed distance. By ordering the buckets by ascending distance, we can prune scanning the buckets once the remaining entries belong to POI that are not close enough. Computing the closest POI along a shortest path is related to the ride sharing algorithm (Section 6.3) but in a different setup.

Our algorithm works on a static graph $G = (V, E)$ as introduced in Section 2.2.1. The set of POI is given as set $X \subseteq V$ of nodes.

6.4.1 POI close to a Node

For a node s and a distance ℓ, we want to compute the set of close POI (Definition 6.9).

Definition 6.9 *Let $X \subseteq V$ be a set of POI. The POI within distance ℓ of a node s is the set $PND(s,\ell) := \{x \in X \mid \mu(s,x) \leq \ell\}$ (**P**OI **N**ode **D**istance).*

For small distance ℓ, a local Dijkstra algorithm can efficiently compute this set. It stops as soon as the minimum key in the priority queue is above ℓ. But Dijkstra's algorithm is inefficient if it settles a lot of non-POI nodes. We propose an algorithm based on the previous batched shortest paths computation techniques, but now we only have a single source s. We precompute for each POI $x \in X$ the backward search space and store it in *backward buckets* $\overleftarrow{\beta}(u)$ at the reached nodes u as described in Section 6.1.1.

At query time, we are given node s and distance ℓ, and we compute $PPD(s,\ell)$ using (6.10).

$$PND(s,\ell) = \left\{ x \;\middle|\; \exists (u, \overrightarrow{\delta}(u)) \in \overrightarrow{\sigma}(s) : \exists (x, \overleftarrow{\delta}_x(u)) \in \overleftarrow{\beta}(u) : \overrightarrow{\delta}(u) + \overleftarrow{\delta}_x(u) \leq \ell \right\} \quad (6.10)$$

Remember that $\overleftarrow{\delta}_x(u)$ denotes the distance from u to x computed by the backward search from x, see Figure 6.27.

Lemma 6.10 *Equation (6.10) is correct.*

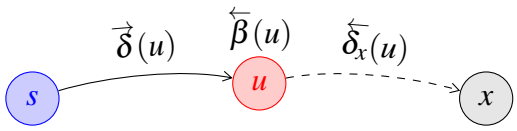


Figure 6.27: Computing the distance from source node s to POI x by scanning the backward bucket at each node u reached by the forward search from s. The path obtained by scanning the backward bucket is represented by a dashed edge, as this path is not actually traversed.

Proof. Let A be the set $PND(s,\ell)$ from Definition 6.9 and B be the set from (6.10). We need to prove that $A = B$.

$\subseteq$: Let $x \in A$. Then $\mu(s,x) \leq \ell$. By (6.6) we know that $\exists(u, \overrightarrow{\delta}(u)) \in \overrightarrow{\sigma}(s)$: $\exists(x, \overleftarrow{\delta_x}(u)) \in \overleftarrow{\beta}(u)$ with $\overrightarrow{\delta}(u) + \overleftarrow{\delta_x}(u) = \mu(s,x) \leq \ell$, and therefore $x \in B$.

$\supseteq$: Let $x \in B$. Then there is a u such that $(u, \overrightarrow{\delta}(u)) \in \overrightarrow{\sigma}(s), (x, \overleftarrow{\delta_x}(u)) \in \overleftarrow{\beta}(u)$ and $\overrightarrow{\delta}(u) + \overleftarrow{\delta_x}(u) \leq \ell$. Again by (6.6) we know that $\overrightarrow{\delta}(u) + \overleftarrow{\delta_x}(u) \geq \mu(s,x)$ and therefore $\mu(s,x) \leq \ell \Rightarrow x \in A$. □

We will store the buckets $\overleftarrow{\beta}(u)$ as arrays, and we sort them ascending by $\overleftarrow{\delta_x}(u)$. This allows to skip scanning a remaining bucket if $\overrightarrow{\delta}(u) + \overleftarrow{\delta_x}(u) > \ell$. The resulting Algorithm 6.3 efficiently computes $PND(s,\ell)$. It computes the forward search space from node s and intersects them with the backward search spaces stored in the buckets. The main part of the algorithm is spent scanning the buckets in Lines 5–7. But this is very fast, as scanning consecutive pieces of main memory is cache-efficient. Also note that we could easily not only compute the set of POI, but also their distance from source node s. We just need to store for each POI the smallest observed distance $\overrightarrow{\delta}(u) + \overleftarrow{\delta_x}(u)$ when we add x to the set of close POI in Line 7.

Algorithm 6.3: PoiNodeDistance(s, ℓ)

input : source node s, distance ℓ
output : $PND(s,\ell)$

```
1 compute σ⃗(s);                                          // forward search space
2 Y := ∅;                                                 // set of close POI
3 foreach (u, δ⃗(u)) ∈ σ⃗(s) ascending by δ⃗(u) do          // loop over reached nodes
4     if δ⃗(u) > ℓ then break;                            // prune remaining nodes
5     foreach (x, δ←x(u)) ∈ β←(u) ascending by δ←x(u) do  // scan bucket
6         if δ⃗(u) + δ←x(u) > ℓ then break;               // prune remaining bucket
7         Y := Y ∪ {x};                                   // node x is close POI
8 return Y
```

6.4.2 POI close to a Path

Definition 6.11 describes the set of POI close to the shortest path between a source node s and a target node t. We deliberately say *the* shortest path although there are potential several shortest paths, as we define a POI to be close in the sense that it requires only a small detour ℓ compared to the shortest-path distance from s to t.

Definition 6.11 *Let $X \subseteq V$ be a set of POI. The POI within detour ℓ between source node s and target node t are the set $PPD(s,t,\ell) := \{x \in X \mid \mu(s,x) + \mu(x,t) \leq \mu(s,t) + \ell\}$ (**P**OI **P**ath **D**istance).*

Computing the POI close to a path is much harder than computing the ones close to a node. Such POI can be far away from the source and target node, and also from a shortest path between them. This problem is similar to Section 6.3, where we want to compute detours. However, here we have a set of POI that we need to match to a path, instead of a set of drivers that needs to be matched to passengers. A naïve algorithm would perform a forward Dijkstra search from s and a backward Dijkstra search from t, both limited to the distance $\mu(s,t) + \ell$. Thus, this search is very expensive, as it has to depend on the shortest-path length between s and t and not only on the detour ℓ.

To engineer a faster algorithm, we compute forward and backward search spaces from each POI x and store their entries into *forward buckets* $\overrightarrow{\beta}(u)$ and *backward buckets* $\overleftarrow{\beta}(u)$ as described in Section 6.1.1. That means that we store additional forward buckets compared to Section 6.4.1. These forward buckets are necessary to compute the distances from the POI to the target node t.

At query time, we are given nodes s, t and distance ℓ, and we compute $PPD(s,\ell)$ using (6.11).

$$PPD(s,t,\ell) = \left\{ x \;\middle|\; \begin{array}{c} \exists(u, \overrightarrow{\delta}(u)) \in \overrightarrow{\sigma}(s) : \exists(x, \overleftarrow{\delta_x}(u)) \in \overleftarrow{\beta}(u), \\ \exists(v, \overleftarrow{\delta}(v)) \in \overleftarrow{\sigma}(t) : \exists(x, \overrightarrow{\delta_x}(v)) \in \overrightarrow{\beta}(v) : \\ \overrightarrow{\delta}(u) + \overleftarrow{\delta_x}(u) + \overrightarrow{\delta_x}(v) + \overleftarrow{\delta}(v) \leq \mu(s,t) + \ell \end{array} \right\} \tag{6.11}$$

Note that $\overleftarrow{\delta_x}(u)$ denotes the distance from u to x computed by the backward search from x, and $\overrightarrow{\delta_x}(v)$ denotes the distance from x to v computed by the forward search from x, see Figure 6.28.

Lemma 6.12 *Equation (6.11) is correct.*

Proof. Let A be the set from the definition of $PPD(s,t,\ell)$ and B be the set from (6.11). We need to prove that $A = B$.

$\subseteq$: Let $x \in A$. Then $\mu(s,x) + \mu(x,t) \leq \mu(s,t) + \ell$. By (6.6) we know that $\exists(u, \overrightarrow{\delta}(u)) \in \overrightarrow{\sigma}(s) : \exists(x, \overleftarrow{\delta_x}(u)) \in \overleftarrow{\beta}(u)$ with $\overrightarrow{\delta}(u) + \overleftarrow{\delta_x}(u) = \mu(s,x)$. And by (6.7) we know that $\exists(v, \overleftarrow{\delta}(v)) \in \overleftarrow{\sigma}(t) : \exists(x, \overrightarrow{\delta_x}(v)) \in \overrightarrow{\beta}(v)$ with $\overrightarrow{\delta_x}(v) + \overleftarrow{\delta}(v) = \mu(x,t)$. Therefore, $x \in B$.

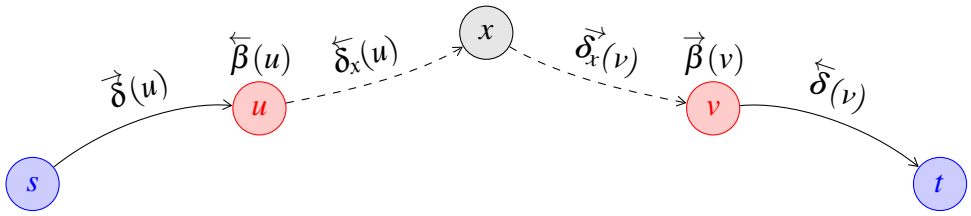


Figure 6.28: Computing the distance via POI x between source node s and target node t by scanning the backward bucket at each node u reached by the forward search from s, and by scanning the forward bucket at each node v reached by the backward search from t. The paths obtained by scanning the buckets are represented by dashed edges, as these paths are not actually traversed.

$\supseteq$: Let $x \in B$. Then there are nodes u, v such that $(u, \overrightarrow{\delta}(u)) \in \overrightarrow{\sigma}(s), (x, \overleftarrow{\delta}_x(u)) \in \overleftarrow{\beta}(u)$, $(v, \overleftarrow{\delta}(v)) \in \overleftarrow{\sigma}(t), (x, \overrightarrow{\delta}_x(v)) \in \overrightarrow{\beta}(v)$ with $\overrightarrow{\delta}(u) + \overleftarrow{\delta}_x(u) + \overrightarrow{\delta}_x(v) + \overleftarrow{\delta}(v) \leq \mu(s,t) + \ell$. Again by (6.6) and (6.7) we know that $\overrightarrow{\delta}(u) + \overleftarrow{\delta}_x(u) \geq \mu(s,x)$, $\overrightarrow{\delta}_x(v) + \overleftarrow{\delta}(v) \geq \mu(x,t)$ and therefore $\mu(s,x) + \mu(x,t) \leq \mu(s,t) + \ell \Rightarrow x \in A$. □

Our Algorithm 6.4 based on (6.11) first computes $PND(s, \mu(s,t) + \ell)$ including the distances from s to the POI. Then it scans the backward buckets of the nodes reached by

Algorithm 6.4: PoiPathDistance(s, t, ℓ)

input : source node s, target node t, distance ℓ
output : $PPD(s,t,\ell)$

```
1  compute σ→(s) and σ←(t);                          // forward and backward search space
2  Y := ∅;                                           // set of close POI
3  d→ := ⟨∞, ..., ∞⟩;                                // tentative distances μ(s,·)
4  ℓ+ := μ(s,t) + ℓ;                                 // maximum path length via POI
5  foreach (u, δ→(u)) ∈ σ→(s) ascending by δ→(u) do  // loop over forward nodes
6      if δ→(u) > ℓ+ then break;                     // prune remaining nodes
7      foreach (x, δ←x(u)) ∈ β←(u) ascending by δ←x(u) do   // scan bucket
8          if δ→(u) + δ←x(u) > ℓ+ then break;        // prune rest of bucket
9          d→[x] := min(d→[x], δ→(u) + δ←x(u));      // update distance from s
10 foreach (u, δ←(u)) ∈ σ←(t) ascending by δ←(u) do  // loop over backward nodes
11     if δ←(u) > ℓ+ then break;                     // prune remaining nodes
12     foreach (x, δ→x(u)) ∈ β→(u) ascending by δ→x(u) do   // scan bucket
13         if δ→x(u) + δ←(u) > ℓ+ then break;        // prune rest of bucket
14         if d→[x] + δ→x(u) + δ←(u) ≤ ℓ+ then Y := Y ∪ {x};   // node x is close POI
15 return Y
```

a backward search form t to obtain distances from the POI to t. If such a distance plus the previously computed distance from s to this POI is within our bound $\mu(s,t)+\ell$, we add the POI to our tentative set of close POI. However, the pruning technique used for ride sharing (Section 6.3.2) cannot be applied, as a POI is a single node and not a path. Computing the detour to each POI in $PPD(s,t,\ell)$ can be done similarly to Algorithm 6.3 by storing the smallest observed detour when we add x to the set of close POI in Line 14.

6.4.3 k-closest POI

The algorithms of Sections 6.4.1 and 6.4.2 compute the POI subject to a distance ℓ. But we can also compute the k-closest POI. To compute them, we modify the previous algorithms. We keep the found POI in a maximum priority queue with their distance. If there are more than k POI in the queue, we remove the one with the largest distance (delete max). The distance ℓ is obtained from the maximum value of the priority queue, once there are k POI in it. As we use ℓ for pruning, we should try to obtain a low value as fast as possible.

POI close to a Node. The k-closest POI are not necessarily unique, as there can be POI with identical distance from the source node s. Formally, we want to compute a set $PNC(s,k)$ of k-closest POI to a node s given by Definition 6.13.

Definition 6.13 *Let $X \subseteq V$ be a set of POI. A set $PNC(s,k)$ (**P**OI **N**ode **C**losest) of k-closest POI to a node s has the properties*

$$PNC(s,k) \subseteq X \tag{6.12}$$

$$|PNC(s,k)| = \min(k, |\{x \in X \mid \mu(s,x) < \infty\}|) \tag{6.13}$$

$$\forall x \in X \setminus PNC(s,k) : \mu(s,x) \geq \max\{\mu(s,y) \mid y \in PNC(s,k)\} \tag{6.14}$$

Our algorithm will return an arbitrary set fulfilling the above properties. However, it requires only small changes to the algorithm to additionally compute all POI with distance of the k^{th} POI. To compute a set $PNC(s,k)$, we modify the previous algorithm (Section 6.4.1). The resulting Algorithm 6.3 efficiently computes $PNC(s,k)$. Note that the maximum priority queue does not only hold the k-closest POI, but also the shortest-path distance from s to t via each POI. So we could additionally return the detours in Line 11 with almost no overhead.

While the correctness of Algorithm 6.3 directly followed from Lemma 6.10, the correctness of Algorithm 6.5 is not so straightforward and we will prove it in Lemma 6.14.

Lemma 6.14 *Algorithm 6.5 computes a set $PNC(s,k)$ that fulfills (6.12), (6.13), and (6.14).*

Algorithm 6.5: PoiNodeClosest(s, k)

input : source node s, cardinality k
output : $PNC(s,k)$

```
1  compute ⃗σ(s);
2  Y := ∅;                                  // maximum priority queue holding the k-closest POI
3  ℓ := ∞;                                                                  // maximum distance
4  foreach (u, ⃗δ(u)) ∈ ⃗σ(s) ascending by ⃗δ(u) do                          // loop over reached nodes
5      if ⃗δ(u) > ℓ then break;                                             // prune remaining nodes
6      foreach (x, ←δ_x(u)) ∈ ←β(u) ascending by ←δ_x(u) do                 // scan bucket
7          if ⃗δ(u) + ←δ_x(u) > ℓ then break;                               // prune rest of bucket
8          Y.update(⃗δ(u) + ←δ_x(u), x);                                    // update distance of POI
9          if |Y| > k then Y.deleteMax();                // remove POI with maximum distance
10         if |Y| = k then ℓ := Y.max();                                    // update maximum detour
11 return {x | (·,x) ∈ Y};                                                 // return k-closest POI
```

Proof. (6.12) holds by the choice of nodes inserted into Y.

Due to the check in Line 9, $|Y| \leq k$. The case $|Y| < k$ can only happen if this check always failed. Thus, $\ell = \infty$ and the algorithm computes all POI reachable from s. So (6.13) also holds.

Let $x \in X \setminus \{x \mid (\cdot,x) \in Y\}$ with $\mu(s,x) < \infty$. By the correctness of Algorithm 6.3 based on Lemma 6.10, we know that all nodes $y \in X$ with $\mu(s,y) < \infty$ and $\mu(s,y) \leq \ell$ have been inserted into Y with distance $\mu(s,y)$. If x has been inserted into Y, we know that $\mu(s,x) \geq \max\{\mu(s,y) = \delta \mid (\delta,y) \in Y\}$ by the definition of a maximum priority queue. Assume that x has not been inserted into Y. Lines 7 and 10 ensure that $\max\{\mu(s,y) = \delta \mid (\delta,y) \in Y\} \leq \ell$. As x has not been inserted, we can follow that $\mu(s,x) > \ell \geq \max\{\mu(s,y) \mid (\cdot,y) \in Y\}$, and (6.14) holds.

POI close to a Path. Formally, we want to compute a set $PPC(s,t,k)$ of k-closest POI to a path given by its source node s and target node t following Definition 6.15.

Definition 6.15 *Let $X \subseteq V$ be a set of POI. A set $PPC(s,t,k)$ (**P**OI **P**ath **C**losest) of k POI within smallest detour between source node s and target node t has the properties*

$$PPC(s,t,k) \subseteq X \tag{6.15}$$

$$|PPC(s,t,k)| = \min(k, |\{x \in X \mid \mu(s,x) + \mu(x,t) < \infty\}|) \tag{6.16}$$

$$\forall x \in X \setminus PPC(s,t,k) : \mu(s,x) + \mu(x,t) \geq \max\{\mu(s,y) + \mu(y,t) \mid y \in PPC(s,t,k)\} \tag{6.17}$$

To compute a set $PPC(s,t,k)$, we modify the previous algorithm (Section 6.4.2). The resulting Algorithm 6.3 efficiently computes $PPC(s,k)$. The observing reader may note that we need more modifications than for Algorithm 6.5 that only computes nodes close to a node. Additionally to adding the maximum priority queue, we also combine the loops over the forward and backward search space. The reason is that when we want to add a POI to the maximum priority queue, we need to have a tentative distance from the source node to the POI, and a tentative distance from the POI to the target. The first distance is computed by looping over the forward search space, the second distance is computed by looping over the backward search space. By combining the loop over forward and backward search space we find some initial POI faster, and once we have k POI, we start pruning by setting ℓ^+ to a finite value in Line 17. However, combining forward and backward search requires to store another array of tentative distances, one for the distances from s to the POI as before, and a new one for the distances from the POI to t. Note again, that the maximum priority queue does not only hold the k-closest POI, but also the shortest-path distance from s to t via each POI.

Algorithm 6.6: PoiPathClosest(s,t,k)

input : source node s, target node t, cardinality k
output : $PPC(s,t,k)$

1 compute $\overrightarrow{\sigma}(s)$ and $\overleftarrow{\sigma}(t)$; // forward and backward search space
2 $Y := \emptyset$; // maximum priority queue holding the k-closest POI
3 $\ell^+ := \infty$; // maximum path length via POI
4 $\overrightarrow{d} := \langle \infty, \ldots, \infty \rangle$; // tentative distances $\mu(s,\cdot)$
5 $\overleftarrow{d} := \langle \infty, \ldots, \infty \rangle$; // tentative distances $\mu(\cdot,t)$
// Merge both search spaces to loop over them by distance.
6 $S := \left\{ (\rightarrow, u, \overrightarrow{\delta}(u)) \mid (u, \overrightarrow{\delta}(u)) \in \overrightarrow{\sigma}(s) \right\} \cup \left\{ (\leftarrow, u, \overleftarrow{\delta}(u)) \mid (u, \overleftarrow{\delta}(u)) \in \overleftarrow{\sigma}(t) \right\}$;
7 **foreach** $(\sim, u, \widetilde{\delta}(u)) \in S$ *ascending by* $\widetilde{\delta}(u)$ **do**
8 **if** $\widetilde{\delta}(u) > \ell^+$ **then break**; // prune remaining nodes
// negation changes direction: $\neg \leftarrow = \rightarrow$ and $\neg \rightarrow = \leftarrow$
9 **foreach** $(x, \overrightarrow{\widetilde{\delta}_x}(u)) \in \overrightarrow{\widetilde{\beta}}(u)$ *ascending by* $\overrightarrow{\widetilde{\delta}_x}(u)$ **do** // scan bucket
10 $d := \widetilde{\delta}(u) + \overrightarrow{\widetilde{\delta}_x}(u)$; // tentative distance from or to POI
11 **if** $d > \ell^+$ **then break**; // prune rest of bucket
12 **if** $\widetilde{d}[x] > d$ **then** // decrease tentative distance
13 $\widetilde{d}[x] := d$; // update distance
14 **if** $\overrightarrow{\widetilde{d}}[x] \neq \infty$ **and** $d + \overrightarrow{\widetilde{d}}[x] \leq \ell^+$ **then** // detour found
15 Y.update$\left(d + \overrightarrow{\widetilde{d}}[x], x\right)$; // update distance of POI
16 **if** $|Y| > k$ **then** Y.deleteMax(); // remove farthest POI
17 **if** $|Y| = k$ **then** $\ell^+ := Y$.max(); // update maximum length
18 **return** $\{x \mid (\cdot, x) \in Y\}$; // return k-closest POI

Lemma 6.16 *Algorithm 6.6 computes a set $PPC(s,t,k)$ that fulfills (6.15), (6.16), and (6.17).*

Proof. (6.15) holds by the choice of nodes inserted into Y.

Due to the check in Line 16, $|Y| \leq k$. The case $|Y| < k$ can only happen if this check always failed. Thus, $\ell^+ = \infty$ and the algorithm computes all POI reachable from s and t. So (6.16) also holds.

Let $x \in X \setminus \{x \mid (\cdot,x) \in Y\}$ with $\mu(s,y) + \mu(y,t) < \infty$. By the correctness of Algorithm 6.4 based on Lemma 6.12, we know that all nodes $y \in X$ with $\mu(s,y) + \mu(y,t) < \infty$ and $\mu(s,y) + \mu(y,t) \leq \ell^+$ have been inserted into Y with distance $\mu(s,y) + \mu(y,t)$. If x has been inserted into Y, we know that $\mu(s,x) + \mu(x,t) \geq \max\{\mu(s,y) + \mu(y,t) = \delta \mid (\delta,y) \in Y\}$ by the definition of a maximum priority queue. Assume that x has not been inserted into Y. Lines 14 and 17 ensure that $\max\{\mu(s,y) + \mu(y,t) = \delta \mid (\delta,y) \in Y\} \leq \ell^+$. As x has not been inserted, we can follow that $\mu(s,x) + \mu(x,t) > \ell^+ \geq \max\{\mu(s,y) + \mu(y,t) \mid y \in PPC(s,t,k)\}$, and (6.17) holds.

6.4.4 Experiments

Instance. Our experiments have been performed on a real-world road network of Germany with 4.4 million nodes and 10.7 million directed edges, provided by PTV AG for scientific use. We do not provide experiments for a larger instance, as the query time for the Dijkstra-based algorithms would be too large.

Environment. Experiments have been done on one core of a single AMD Opteron Processor 270 clocked at 2.0 GHz with 8 GiB main memory and 2×1 MiB L2 cache, running SuSE Linux 11.1 (kernel 2.6.27). The program was compiled by the GNU C++ compiler 4.3.2 using optimization level 3. We use CH (aggressive approach [59]) as bidirected and non-goaldirected speed-up technique.

Basic setup. The numbers are averages over 1 000 queries. We report results for different fractions of POI. The POI have been picked uniformly at random. For the POI computation close to a node, we pick the node uniformly at random. For the POI computation close to a shortest path, we either pick the source and target node uniformly at random, and report results for different fractions of POI. Or we pick source and target node with a certain shortest-path distance and 1% POI. We select them by repeatedly picking a source node uniformly at random, running Dijkstra's algorithm, and taking the first settled node with distance larger or equal to the desired shortest-path distance as target.

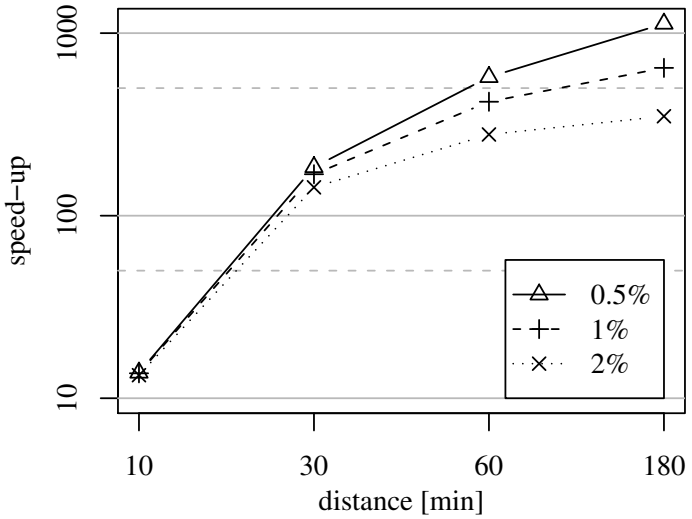


Figure 6.29: Time speed-up of our new algorithm to compute POI close to a node. Each line corresponds to a fixed fraction of POI. The vertical axis has logarithmic scale.

POI close to a Node. In Table 6.30, we compare the performance of a simple algorithm based on Dijkstra's algorithm to our new algorithm. As expected, the runtime of Dijkstra's algorithm only depends on distance ℓ and not on the fraction of POI. That is because it settles all nodes within distance ℓ independent of their belonging to the set of POI. Its performance is only good to find very close POI within 10–30 minutes. Beyond that, the computational time is too high, as the increases is quadratic in ℓ. But even for 10 minutes distance, our new algorithm has a speed-up of more than 10, see also Figure 6.29. Also, our new algorithm depends much less on the distance ℓ, even the computation of all POI within 3 hours is done within milliseconds. However, the query time also depends on the fraction of POI, as with more POI, there are more bucket entries to scan. So the speed-up of our algorithm increases with increasing distance ℓ, but as expected decreases with the fraction of POI.

Table 6.30: Performance of algorithms that compute all POI close to a node.

(a) comparison

		time [ms] distance ℓ [min] =				settled nodes [$\times 10^3$] distance ℓ [min] =			
method	POI	10	30	60	180	10	30	60	180
Dijkstra	0.5%	1.4	26	147	1 588	2.4	41	201	1 807
	1%	1.4	26	147	1 587	2.4	41	201	1 807
	2%	1.4	26	147	1 589	2.4	41	201	1 807
CH + buckets	0.5%	0.10	0.14	0.26	1.41	0.02	0.07	0.12	0.26
	1%	0.10	0.16	0.35	2.46	0.02	0.07	0.12	0.26
	2%	0.10	0.18	0.53	4.53	0.02	0.07	0.12	0.26

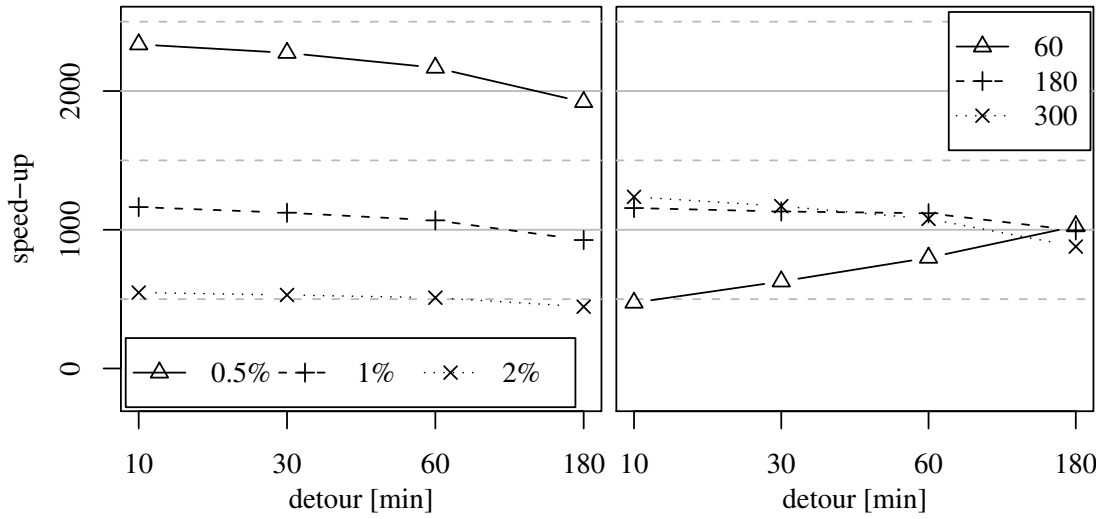
(b) time speed-up

fraction	distance ℓ [min] =			
of POI	10	30	60	180
0.5%	14	185	575	1 127
1%	14	169	421	645
2%	13	143	278	350

(c) scanned bucket entries [$\times 10^3$]

fraction	distance ℓ [min] =			
of POI	10	30	60	180
0.5%	0.05	1.2	8.8	113
1%	0.09	2.4	17.6	226
2%	0.16	4.8	35.0	452

POI close to a Path. Computing the POI with small detour compared to the shortest-path distance between a source and target node is much harder than computing the POI close to a node, see Table 6.32. This is because we need for each POI the shortest-path distance from the source and to the target of the path, and thus we cannot prune the Dijkstra with the detour distance but also need to add the shortest-path distance. Therefore, the runtime of a Dijkstra based algorithm is quite large, and increases with path length and detour, but again is independent of the number of POI. Our new algorithm is much faster, but the runtime increases with detour, number of POI, and path length. For random queries, speed-up does not increase much with increasing detour, but is already very high for small detours, see also Figure 6.31. The shortest-path distance of a random source target pair is around 209 minutes, and therefore already very long. We see that for shorter paths, the speed-up still increases with the detour. However, for very long paths plus long detours, the speed-up of our algorithm slightly decreases as we then need to scan over almost all bucket entries.



(a) fraction of POI (b) path [min]

Figure 6.31: Time speed-up of our new algorithm to compute POI close to a path. The vertical axis has non-logarithmic scale.

Table 6.32: Performance of algorithms that compute all POI with small detour to a path.

(a) comparison

method	fraction of POI	time [ms] detour ℓ [min] = 10	30	60	180	settled nodes [$\times 10^3$] detour ℓ [min] = 10	30	60	180
Dijkstra	0.5%	4 628	5 319	6 185	8 421	4 741	5 332	6 134	8 154
	1%	4 686	5 359	6 254	8 409	4 741	5 332	6 134	8 154
	2%	4 664	5 355	6 261	8 428	4 741	5 332	6 134	8 154
CH + buckets	0.5%	2.0	2.3	2.9	4.4	0.34	0.34	0.34	0.34
	1%	4.0	4.8	5.9	9.1	0.34	0.34	0.34	0.34
	2%	8.5	10.1	12.3	18.9	0.34	0.34	0.34	0.34
	path [min]								
Dijkstra	60	399	737	1 432	5 328	575	965	1 719	5 624
	180	3 833	4 609	5 789	8 661	4 048	4 754	5 777	8 337
	300	7 357	7 900	8 485	9 215	7 229	7 663	8 121	8 731
CH + buckets	60	0.8	1.2	1.8	5.2	0.34	0.34	0.34	0.34
	180	3.3	4.1	5.2	8.7	0.34	0.34	0.34	0.34
	300	5.9	6.8	7.9	10.5	0.34	0.34	0.34	0.34

(b) time speed-up

fraction of POI	detour ℓ [min] = 10	30	60	180
0.5%	2 336	2 275	2 168	1 921
1%	1 164	1 123	1 068	926
2%	547	530	510	445
path [min]				
60	474	627	798	1 025
180	1 157	1 132	1 119	996
300	1 237	1 170	1 079	879

(c) scanned bucket entries [$\times 10^3$]

fraction of POI	detour ℓ [min] = 10	30	60	180
0.5%	176	203	243	378
1%	353	406	486	757
2%	707	813	974	1 515
path [min]				
60	33	57	104	404
180	269	327	418	736
300	565	617	689	869

***k*-closest POI to a Node.** In Table 6.33, we compare the performance of a simple algorithm based on Dijkstra's algorithm to our new algorithm. The simple algorithm stops as soon as k POI are found, and therefore its query time increases with k. Also, as the expected number of settled nodes until k POI are found depends on the fraction of POI in the graph, the query time decreases with increasing fraction of POI. Due to the nature of Dijkstra's algorithm, it finishes once k POI have been settled. In contrary, our new algorithm almost only depends on the number k and is independent of the fraction of POI in the graph. That is because the most time is used scanning the buckets, and we scan an almost constant number of bucket entries per found POI. So we achieve best speed-up with a small fraction of POI and a large k, see also Figure 6.34. Also note that the number of scanned bucket entries increases superlinearly with k. That is because with a larger k, we need to consider a lot of bucket entries that give a suboptimal distance from source node to POI. Almost any scanned bucket entry gives a distance to a POI that needs to be considered, and we need to access the maximum priority queue holding the k closest POI for it.

Table 6.33: Performance of algorithms that compute the k-closest POI to a node. POI that are not pruned by the maximum detour of the maximum priority queue are *considered*. A POI can be considered multiple times for entries in different buckets.

(a) comparison

		time [ms]				settled nodes [$\times 10^3$]				POI considerations			
	fraction	$k =$				$k =$				$k =$			
method	of POI	10	100	1 k	10 k	10	100	1 k	10 k	10	100	1 k	10 k
Dijkstra	0.5%	1.2	11.1	133	1 662	2.0	20.0	201	2 001	10	100	1.00 k	10 k
	1%	0.6	5.4	62	786	1.0	10.0	100	1 002	10	100	1.00 k	10 k
	2%	0.4	2.8	29	368	0.5	5.0	50	500	10	100	1.00 k	10 k
CH + buckets	0.5%	0.10	0.14	0.49	4.61	0.03	0.06	0.13	0.27	35	545	9.22 k	141 k
	1%	0.09	0.13	0.48	4.71	0.02	0.04	0.10	0.22	32	472	7.92 k	126 k
	2%	0.09	0.13	0.46	4.70	0.02	0.03	0.08	0.17	29	418	6.74 k	111 k

(b) time speed-up

fraction	$k =$			
of POI	10	100	1 k	10 k
0.5%	12	79	271	361
1%	7	41	130	167
2%	4	21	63	78

(c) scanned bucket entries

fraction	$k =$			
of POI	10	100	1 k	10 k
0.5%	46	575	9.29 k	141 k
1%	41	495	7.98 k	126 k
2%	36	436	6.78 k	111 k

***k*-closest POI to a Path.** In Table 6.36, we compare the performance of a simple algorithm based on a bidirectional Dijkstra algorithm to our new algorithm. Even computing just the 10 closest POI takes several seconds with the simple algorithm. The query time increases with increasing k, and also with decreasing fraction of POI, as we need to settle

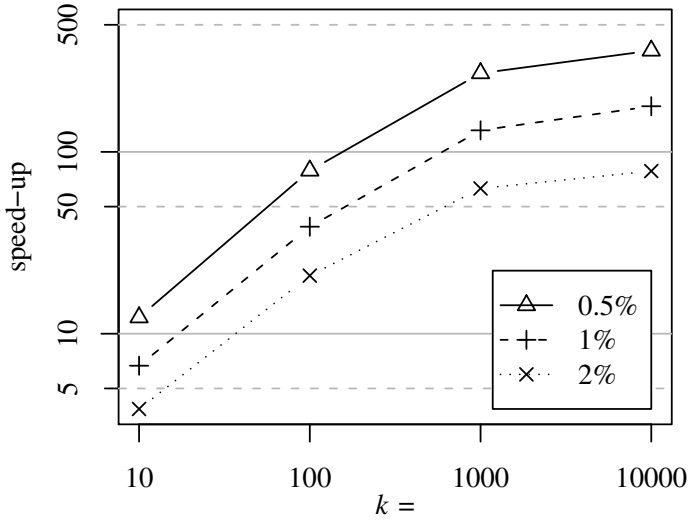


Figure 6.34: Time speed-up of our new algorithm to compute the k-closest POI to a node. Each line corresponds to a fixed fraction of POI. Both axis have logarithmic scale.

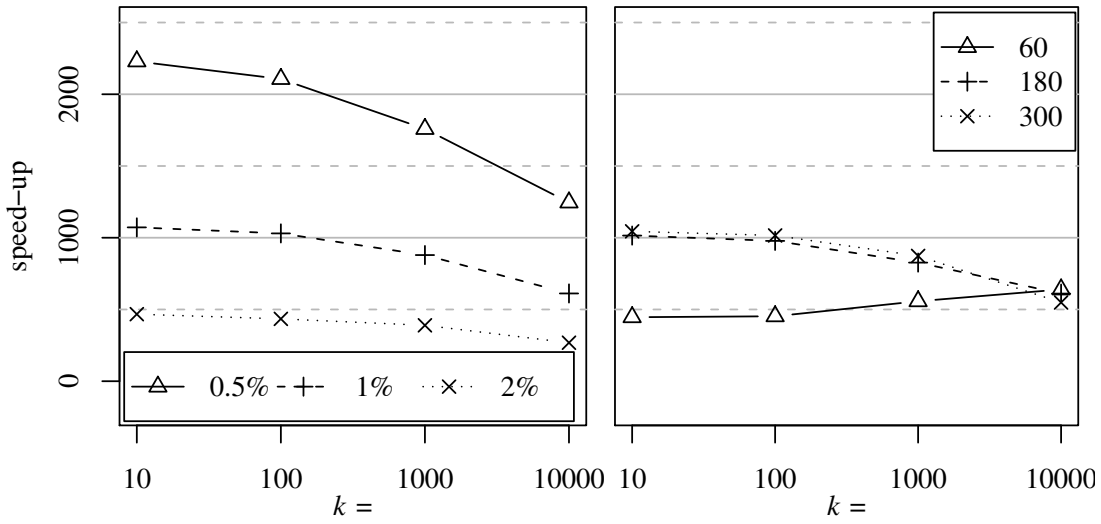


Figure 6.35: Time speed-up of our new algorithm to compute the k-closest POI to a path. The vertical axis has non-logarithmic scale.

more nodes. Our new algorithm has an increasing query time with k, but also with the fraction of POI, as the buckets contain more entries. Interestingly, a lot of bucket entries get pruned by the maximum distance of our maximum priority queue. This is different than for the k-closest POI to a node, where almost no bucket entry got pruned. The speed-up of our algorithm drops significantly for larger k, arguably due to cache-effects as the maximum priority queue is large and gets accessed a lot. But the speed-up still remains significantly above 500, see also Figure 6.35.

Table 6.36: Performance of algorithms that compute the k-closest POI to a path. POI that are not pruned by the maximum detour of the maximum priority queue are *considered*. A POI can be considered multiple times for entries in different buckets.

(a) comparison

		time [ms]				settled nodes [$\times 10^3$]				POI considerations			
	fract.	k =				k =				k =			
method	of POI	10	100	1 k	10 k	10	100	1 k	10 k	10	100	1 k	10 k
Dijkstra	0.5%	4 379	4 566	5 356	8 878	4 502	4 667	5 359	8 533	44	334	2.16 k	12.2 k
	1%	4 351	4 481	4 940	7 587	4 472	4 602	5 006	7 339	46	369	2.50 k	14.5 k
	2%	4 320	4 440	4 727	6 340	4 451	4 552	4 805	6 223	47	403	2.87 k	17.3 k
CH + buckets	0.5%	1.96	2.17	3.05	7.13	0.34	0.34	0.34	0.34	121	768	4.72 k	29.8 k
	1%	4.06	4.35	5.62	12.41	0.34	0.34	0.34	0.34	137	892	5.40 k	34.2 k
	2%	9.25	10.22	12.14	23.64	0.34	0.34	0.34	0.34	154	1 029	6.30 k	39.5 k
path [min]													
Dijkstra	60	290	372	1 029	6 331	443	544	1 297	6 595	34	219	1.31 k	10.5 k
	180	3 472	3 658	4 169	7 303	3 728	3 880	4 327	7 059	46	368	2.39 k	13.2 k
	300	7 038	7 159	7 486	8 524	6 991	7 092	7 315	8 113	49	422	3.07 k	17.5 k
CH + buckets	60	0.65	0.82	1.85	9.90	0.34	0.34	0.34	0.34	73	460	3.22 k	28.1 k
	180	3.42	3.74	5.04	11.96	0.34	0.34	0.34	0.34	132	833	4.98 k	31.5 k
	300	6.74	7.05	8.57	15.54	0.34	0.34	0.34	0.34	160	1 077	6.46 k	38.2 k

(b) time speed-up

fraction	k =			
of POI	10	100	1 000	10 000
0.5%	2 229	2 107	1 758	1 246
1%	1 072	1 030	879	612
2%	467	434	390	268
path [min]				
60	446	453	557	640
180	1 015	978	827	611
300	1 044	1 016	873	549

(c) scanned bucket entries

fraction	k =			
of POI	10	100	1 k	10 k
0.5%	171 k	180 k	213 k	399 k
1%	340 k	353 k	394 k	625 k
2%	676 k	696 k	749 k	1 017 k
path [min]				
60	27 k	35 k	89 k	525 k
180	254 k	268 k	314 k	581 k
300	547 k	560 k	590 k	704 k

Precomputation. The precomputation of the buckets takes time and space roughly linear in the number of POI, as you can see in Table 6.37. Per 10k POI on the German road network, the precomputation takes about 2 seconds and requires about 7 MiB per direction. Computing the POI close to a node requires only backward buckets, as we are only interested in the distance to the POI. But to compute POI close to paths, we require also forward buckets, thus doubling the precomputation effort. Note that the precomputation can be easily parallelized.

Table 6.37: Precomputation performance.

		close to a node			close to a path		
fraction	POI	time	bucket entries		time	bucket entries	
of POI	$[\times 10^3]$	[s]	$[\times 10^6]$	[MiB]	[s]	$[\times 10^6]$	[MiB]
0.5%	22	4.1	1.9	14	7.4	3.9	29
1%	44	8.0	3.8	29	14.7	7.7	59
2%	88	15.9	7.6	58	29.3	15.4	118

6.5 Concluding Remarks

Review. The basic idea of precomputing and storing search spaces is useful to efficiently solve a variety of problems. We used the idea to present new efficient algorithm for three different problems.

The first problem is the computation of a time-dependent travel time table. We present five different algorithms with different trade-offs between preprocessing time, space and query time. An important ingredient is the efficient exact intersection of forward and backward search spaces using approximate TTFs. The computation of additional data speeds up the queries, but usually increases precomputation time and space. A large impact on the precomputation time and space has the computation of approximate instead of exact TTFs. We are able to reduce time by more than one and space by more than two orders of magnitude with an average error of less than 1%. Furthermore, we provide theoretical error bounds and showed that these bounds are also applicable for TCH queries on approximate TTFs.

The second contribution in this chapter is an algorithmic solution to efficiently compute detours to match ride sharing offers and request. Our algorithm is the first one feasible in practice, even for large datasets, with matching times of a few milliseconds. As a match with small detour is potentially far away from source and target, basic pruning techniques fail. Still, we are able to efficiently prune the computation by additionally computing search spaces in the other direction than required to compute the detours.

Finally, we present a new simple but efficient algorithm to compute POI close to a node or the shortest path of a pair of nodes. The computation of POI in a very small radius of ten minutes is accelerated by more than one order of magnitude compared to Dijkstra's algorithm. For larger radii, the speed-up increases to three orders of magnitude. The computation of POI close to the shortest path benefits the most from our new technique, it takes now milliseconds instead of seconds.

Future Work. Our time-dependent algorithm is an important step to a time-dependent transit node routing algorithm [14]. Transit node routing is currently among the fastest speedup techniques for time-independent road networks and essentially reduces the shortest path search to a few table look-ups. Our algorithms can either compute or completely replace such tables. The algorithms to the other two problems are currently not time-dependent, it would be interesting to adapt them to the time-dependent scenario. However, equally interesting would be an adaption to flexible scenarios (Chapter 5).

Additionally, the ride sharing problem contains an abundance of interesting open problems. Incorporating car switching and multiple passengers per car will bring new and interesting algorithmic challenges. An adaption of the algorithm to the shared taxi system of developing countries will be very interesting as well.

We see that precomputing search spaces is an efficient concept to solve application-oriented problems. It would be interesting to see what other problems can be addressed

in addition to the ones presented in this thesis, especially with regard to nontrivial enhancements such as the pruning used for ride sharing.

A new CH-based algorithm for one-to-all computation on GPUs [40] is able to compute the shortest-path distances within a few milliseconds on continental-sized road networks. It can be used when the buckets grow too large. Our ride sharing algorithm is sufficiently fast when dealing with 100 000 offers, but for more offers, this GPU-based algorithm can be used to compute the detours, the idea of detour computation stays the same. The computation of close POI can also benefit from this new GPU-based algorithm in case that POI within a very large distance should be computed, or there are a lot of POI in the graph. However, most commonly, the distances are small and the fraction of POI in the graph is low, and our algorithm will be faster and more resource-efficient. Furthermore, compared to the GPU-based algorithm, our algorithm allows a mobile implementation based on mobile CH [126].

Also very recent results [1] show that the search spaces of CH can be additionally downsized. This downsizing can directly increase the performance of all algorithms presented in this chapter, especially when a large number of search spaces is stored.

References. Section 6.2 is based on a conference paper [66] which the author of this thesis published together with Peter Sanders. Section 6.3 is based on a technical report [63] and a conference paper [64] published together with Dennis Luxen, Sabine Neubauer, Peter Sanders, and Lars Volker. Section 6.4 has not been published before. Some wordings of these articles are used in this thesis.

7

Discussion

7.1 Conclusion

Advanced route planning is a vast field of research. While some of its problems can be solved by just applying basic route planning algorithms, other problems impose significant challenges. We addressed some of these challenges and presented the most efficient algorithms in the areas of *public transportation*, *flexible queries*, and *batched shortest-path computation*. These algorithms are significantly more advanced than basic route planning algorithms, as they require new models, new algorithmic ingredients, or even completely new algorithmic ideas. All of our algorithms were designed, implemented and evaluated following the paradigm of *algorithm engineering*.

Our *fully realistic public transportation* routing algorithm is the first and only one able to route efficiently in very large and poorly structured networks. Based on the concept of *transfer patterns* it allows a fast and scalable query algorithm. Precomputing the transfer patterns is done using a hierarchical hub-station approach, that is augmented by heuristic ingredients to cope with the irregularities contained in every realistic public transportation network. The resulting algorithm is used on a major public transportation website. Furthermore, we showed how to efficiently contract public transportation networks in the scenario with realistic transfer durations using a new station graph model.

The adaption of Dijkstra's algorithm or A^* search to flexible queries on road networks is rather simple as no precomputation is performed. However, *with precomputation*, major algorithmic augmentations are required to support flexible queries. We augmented the concept of node contraction to the scenarios with two edge weights and edge restrictions. An important ingredient is to store information with shortcuts to prune their relaxation depending on the current query parameters. Augmenting ALT to flexible scenarios required the computation of feasible lower bounds. To tighten the lower bounds in dependence of the query parameters, we additionally computed landmarks and distances for some selected query parameters. The resulting algorithms achieve a speed-up of over three orders of magnitude over Dijkstra's algorithm, with preprocessing time of a few hours on continental-sized road networks.

Besides that, we looked at several problems requiring batched shortest-path computations. We show a new algorithm for time-dependent travel time computation that efficiently uses approximations. Instead of computing a whole travel time table, we provide an interface to such a table and present various algorithms to implement this interface. One of these algorithms has precomputation complexity linear in the number of rows plus columns of the table, and constant query time. Also, we are the first to provide approximation guarantees for time-dependent speed-up techniques by successfully addressing the problem of stacked errors of chained travel time functions. Furthermore, we generalize the idea of search space precomputation to develop new algorithmic approaches to ride sharing and point-of-interest location. We efficiently find matches with small detour, without relying on restricted common databases or spatial data structures.

7.2 Future Work

We summarize the future work discussed at the end of Chapters 4–6. The transfer patterns approach for routing in public transportation networks is very successful except for its very high precomputation time. It would be desirable to further reduce this time, and maybe even perform an *exact* precomputation. This requires new ideas to address the irregularities of public transportation networks. Also, the computed transfer patterns can be used to propose some good connections independent of the departure time.

Our flexible scenarios should be combined into a single algorithm, and also extended to further scenarios such as time-dependency. Then, they can be used to augment algorithms that where previously designed for basic speed-up techniques, especially when relying on node contraction. For example, we could use them to augment our batched shortest-path algorithms to flexible scenarios.

The batched shortest-paths algorithms presented in this theses follow the concept of search space precomputation. It would be interesting to find more applications where this concept can be applied, especially when nontrivial enhancements are required. Also, the time-dependent travel time table algorithm can be used to build a time-dependent transit node routing algorithm to further speed up arbitrary time-dependent one-to-one queries.

7.3 Outlook

In the long run, our goal should be to build the *ultimate route planning system*. All the different aspects of route planning should be combined into a single flexible, multi-modal, multi-criteria, dynamic, and time-dependent system. It should be able to efficiently answer single shortest-path queries, batched shortest-path queries, and even be able to compute alternative routes. We think this thesis is an important step towards such an ultimate route planner, but we also know that there is still a long way to go. And most likely, we have to make some *clever compromises* along the way. We should not only concentrate on developing efficient algorithms to a given problem, but reconsider

the problem definition as well. Of course, redefining the problem must not give the idea behind the initial problem definition away. The resulting algorithms must still produce good practical results. This pragmatic philosophy to solve problems therefore focuses on the usability of the algorithms in practice, and not necessarily on exact mathematical definitions. This allows us to trade some functionality for more efficient algorithms. In this thesis, we already made such trade-offs, namely the heuristic transfer patterns computation in Section 4.2, and the linear combination of multiple edge weights instead of Pareto-optimality in Section 5.2. Although, from an academic viewpoint, it would be desirable not make such compromises, they often have indispensable advantages in practice. We showed that in both cases, our resulting algorithms became significantly more efficient, without really affecting the quality of the query results. Nevertheless, we should always strive to avoid as many compromises as possible. Recent history shows us that this can lead to even more efficient algorithms: Todays fastest speed-up techniques for basic route planning in road networks are all *exact*.

As important intermediate goals towards the ultimate route planning system, we identify

- the creation of an ultimate route planning system restricted to *road networks*,
- a fully realistic route planning algorithm with fast precomputation for *public transportation*, for example by enhancing the transfer patterns precomputation (Section 4.2), and
- developing efficient approaches to *multi-modal* problems.

Especially multi-modal problems are difficult, as not only efficient algorithms are missing, but also formal definitions of the desired solutions that are efficiently computable. The only efficient algorithm we are aware of just works for special combinations, such as the combination of walking and (not fully realistic) public transportation [42, 119]. But for interesting cases like the combination of car and public transportation, no efficient algorithms are known.

Bibliography

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

[2] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. Alternative Routes in Road Networks. In Festa [56], pages 23–34.

[3] Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. Highway Dimension, Shortest Paths, and Provably Efficient Algorithms. In Moses Charikar, editor, *Proceedings of the 21st Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'10)*, pages 782–793. SIAM, 2010.

[4] *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*. SIAM, 2007.

[5] Jacob M. Anthonisse. The rush in a directed graph. Technical Report BN 9/71, Stichting Mathematisch Centrum, 2e Boerhaavestraat 49 Amsterdam, Oct 1971.

[6] *Proceedings of the 3rd Workshop on Algorithmic Methods and Models for Optimization of Railways (ATMOS'03)*, volume 92 of *Electronic Notes in Theoretical Computer Science*, 2004.

[7] *Proceedings of the 10th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'10)*, OpenAccess Series in Informatics (OASIcs), 2010.

[8] Yossi Azar, Y. Bartal, E. Feuerstein, Amos Fiat, Stefano Leonardi, and A. Rosen. On Capital Investment. *Algorithmica*, 25(1):22–36, 1999.

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

[10] Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. Fast Routing in Very Large Public Transportation Networks using Transfer Patterns. In *Proceedings of the 18th Annual European Symposium on Algorithms (ESA'10)*, Lecture Notes in Computer Science, pages 290–301. Springer, 2010.

[11] Holger Bast, Stefan Funke, and Domagoj Matijevic. TRANSIT - Ultrafast Shortest-Path Queries with Linear-Time Preprocessing. In Demetrescu et al. [48].

[12] Holger Bast, Stefan Funke, and Domagoj Matijevic. Ultrafast Shortest-Path Queries via Transit Nodes. In Demetrescu et al. [49], pages 175–192.

[13] Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes. In Transit to Constant Shortest-Path Queries in Road Networks. In ALENEX'07 [4], pages 46–59.

[14] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. Fast Routing in Road Networks with Transit Nodes. *Science*, 316(5824):566, 2007.

[15] Gernot Veit Batz, Daniel Delling, Peter Sanders, and Christian Vetter. Time-Dependent Contraction Hierarchies. In *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX'09)*, pages 97–105. SIAM, April 2009.

[16] Gernot Veit Batz, Robert Geisberger, Sabine Neubauer, and Peter Sanders. Time-Dependent Contraction Hierarchies and Approximation. In Festa [56], pages 166–177.

[17] Gernot Veit Batz, Robert Geisberger, and Peter Sanders. Time Dependent Contraction Hierarchies - Basic Algorithmic Ideas. Technical report, ITI Sanders, Faculty of Informatics, Universität Karlsruhe (TH), 2008.

[18] Gernot Veit Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. Minimum Time-Dependent Travel Times with Contraction Hierarchies. Submitted to JEA, 2011.

[19] Reinhard Bauer, Tobias Columbus, Bastian Katz, Marcus Krug, and Dorothea Wagner. Preprocessing Speed-Up Techniques is Hard. In *Proceedings of the 7th Conference on Algorithms and Complexity (CIAC'10)*, volume 6078 of *Lecture Notes in Computer Science*, pages 359–370. Springer, 2010.

[20] Reinhard Bauer and Daniel Delling. SHARC: Fast and Robust Unidirectional Routing. In Ian Munro and Dorothea Wagner, editors, *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX'08)*, pages 13–26. SIAM, April 2008.

[21] Reinhard Bauer and Daniel Delling. SHARC: Fast and Robust Unidirectional Routing. *ACM Journal of Experimental Algorithmics*, 14(2.4):1–29, August 2009. Special Section on Selected Papers from ALENEX 2008.

[22] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. In McGeoch [104], pages 303–318.

[23] 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, January 2010. Special Section devoted to WEA'08.

[24] Richard Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[25] Annabell Berger, Daniel Delling, Andreas Gebhardt, and Matthias Müller–Hannemann. Accelerating Time-Dependent Multi-Criteria Timetable Information is Harder Than Expected. In *Proceedings of the 9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'09)*, Dagstuhl Seminar Proceedings, 2009.

[26] Annabell Berger and Matthias Müller–Hannemann. Subpath-Optimality of Multi-Criteria Shortest Paths in Time- and Event-Dependent Networks. Technical Report 1, University Halle-Wittenberg, Institute of Computer Science, 2009.

[27] Olli Bräysy and Michel Gendreau. Vehicle Routing Problem with Time Windows, Part I: Route Construction and Local Search Algorithms. *Transportation Science*, 39(1):104–118, February 2005.

[28] Olli Bräysy and Michel Gendreau. Vehicle Routing Problem with Time Windows, Part II: Metaheuristics. *Transportation Science*, 39(1):119–139, February 2005.

[29] Gerth Brodal and Riko Jacob. Time-dependent Networks as Models to Achieve Fast Exact Time-table Queries. In ATMOS'03 [6], pages 3–15.

[30] Edith Brunel, Daniel Delling, Andreas Gemsa, and Dorothea Wagner. Space-Efficient SHARC-Routing. In Festa [56], pages 47–58.

[31] Tom Caldwell. On Finding Minimum Routes in a Network With Turn Penalties. *Communications of the ACM*, 4(2), 1961.

[32] Tobias Columbus. On the Complexity of Contraction Hierarchies, 2009. Student's thesis - Karlsruhe Institute of Technology - ITI Wagner.

[33] K. Cooke and E. Halsey. The Shortest Route Through a Network with Time-Dependent Intermodal Transit Times. *Journal of Mathematical Analysis and Applications*, 14(3):493–498, 1966.

[34] Joseph C. Culberson and Jonathan Schaeffer. Pattern Databases. *Computational Intelligence*, 14(3):318–334, 1998.

[35] George B. Dantzig. *Linear Programming and Extensions*. Princeton University Press, 1962.

[36] Jonathan Dees. Computing Alternative Routes in Road Networks. Master's thesis, Karlsruhe Institut für Technologie, Fakultät für Informatik, April 2010.

[37] Daniel Delling. Time-Dependent SHARC-Routing. In ESA'08 [55], pages 332–343. Best Student Paper Award - ESA Track B.

[38] Daniel Delling. *Engineering and Augmenting Route Planning Algorithms*. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2009.

[39] Daniel Delling. Time-Dependent SHARC-Routing. *Algorithmica*, July 2009. Special Issue: European Symposium on Algorithms 2008.

[40] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. PHAST: Hardware-Accelerated Shortest Path Trees. Technical Report MSR-TR-2010-125, Microsoft Research, 2010.

[41] Daniel Delling and Giacomo Nannicini. Bidirectional Core-Based Routing in Dynamic Time-Dependent Road Networks. In Seok-Hee Hong, Hiroshi Nagamochi, and Takuro Fukunaga, editors, *Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC'08)*, volume 5369 of *Lecture Notes in Computer Science*, pages 813–824. Springer, December 2008.

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

[43] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering Route Planning Algorithms. In Jürgen Lerner, Dorothea Wagner, and Katharina A. Zweig, editors, *Algorithmics of Large and Complex Networks*, volume 5515 of *Lecture Notes in Computer Science*, pages 117–139. Springer, 2009.

[44] Daniel Delling and Dorothea Wagner. Landmark-Based Routing in Dynamic Graphs. In Demetrescu [47], pages 52–65.

[45] Daniel Delling and Dorothea Wagner. Pareto Paths with SHARC. In Jan Vahrenhold, editor, *Proceedings of the 8th International Symposium on Experimental Algorithms (SEA'09)*, volume 5526 of *Lecture Notes in Computer Science*, pages 125–136. Springer, June 2009.

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

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

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

[49] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*. American Mathematical Society, 2009.

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

[51] John F. Dillenburg, Ouri Wolfson, and Peter C. Nelson. The Intelligent Travel Assistant. In *ITSS 2002: Proceedings of the 5h International Conference on Intelligent Transportation Systems*, pages 691–696. IEEE Computer Society, September 2002.

[52] Yann Disser, Matthias Müller–Hannemann, and Mathias Schnee. Multi-Criteria Shortest Paths in Time-Dependent Train Networks. In McGeoch [104], pages 347–361.

[53] Alberto V. Donati, Roberto Montemanni, Norman Casagrande, Andrea E. Rizzoli, and Luca M. Gambardella. Time dependent vehicle routing problem with a multi ant colony system. *European Journal of Operational Research*, 185:1174–1191, 2008.

[54] Stuart E. Dreyfus. An Appraisal of Some Shortest-Path Algorithms. *Operations Research*, 17(3):395–412, 1969.

[55] *Proceedings of the 16th Annual European Symposium on Algorithms (ESA'08)*, volume 5193 of *Lecture Notes in Computer Science*. Springer, September 2008.

[56] Paola Festa, editor. *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*, volume 6049 of *Lecture Notes in Computer Science*. Springer, May 2010.

[57] Lester R. Ford, Jr. Network Flow Theory. Technical Report P-923, Rand Corporation, Santa Monica, California, 1956.

[58] Linton Clarke Freeman. A Set of Measures of Centrality Based Upon Betweeness. *Sociometry*, 40:35–41, 1977.

[59] Robert Geisberger. Contraction Hierarchies. Master's thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2008. `http://algo2.iti.uni-karlsruhe.de/documents/routeplanning/geisberger_dipl.pdf`.

[60] Robert Geisberger. Contraction of Timetable Networks with Realistic Transfers. Technical report, ITI Sanders, Faculty of Informatics, Universität Karlsruhe (TH), 2009.

[61] Robert Geisberger. Contraction of Timetable Networks with Realistic Transfers. In Festa [56], pages 71–82.

[62] Robert Geisberger, Moritz Kobitzsch, and Peter Sanders. Route Planning with Flexible Objective Functions. In *Proceedings of the 12th Workshop on Algorithm Engineering and Experiments (ALENEX'10)*, pages 124–137. SIAM, 2010.

[63] Robert Geisberger, Dennis Luxen, Sabine Neubauer, Peter Sanders, and Lars Volker. Fast Detour Computation for Ride Sharing. Technical report, ITI Sanders, Faculty of Informatics, Universität Karlsruhe (TH), 2009.

[64] Robert Geisberger, Dennis Luxen, Peter Sanders, Sabine Neubauer, and Lars Volker. Fast Detour Computation for Ride Sharing. In ATMOS'10 [7].

[65] Robert Geisberger, Michael Rice, Peter Sanders, and Vassilis Tsotras. Route Planning with Flexible Edge Restrictions. Submitted to JEA, 2011.

[66] Robert Geisberger and Peter Sanders. Engineering Time-Dependent Many-to-Many Shortest Paths Computation. In ATMOS'10 [7].

[67] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In McGeoch [104], pages 319–333.

[68] Frank Geraets, Leo G. Kroon, Anita Schöbel, Dorothea Wagner, and Christos Zaroliagis. *Algorithmic Methods for Railway Optimization*, volume 4359 of *Lecture Notes in Computer Science*. Springer, 2007.

[69] Andrew V. Goldberg and Chris Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 156–165. SIAM, 2005.

[70] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. In *Proceedings of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX'06)*, pages 129–143. SIAM, 2006.

[71] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Better Landmarks Within Reach. In Demetrescu [47], pages 38–51.

[72] Andrew V. Goldberg and Renato F. Werneck. Computing Point-to-Point Shortest Paths from External Memory. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05)*, pages 26–40. SIAM, 2005.

[73] Ronald J. Gutman. Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX'04)*, pages 100–111. SIAM, 2004.

[74] Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57. ACM Press, 1984.

[75] Horst W. Hamacher, Stefan Ruzika, and Stevanus A. Tjandra. Algorithms for time-dependent bicriteria shortest path problems. *Discrete Optimization*, 3(3):238–254, September 2006.

[76] P. Hansen. Bricriteria Path Problems. In Günter Fandel and T. Gal, editors, *Multiple Criteria Decision Making – Theory and Application –*, pages 109–127. Springer, 1979.

[77] Peter E. Hart, Nils Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.

[78] Stephan Hartwig and Michael Buchmann. Empty Seats Travelling. Technical report, Nokia Research Center, 2007.

[79] Hideki Hashimoto, Mutsunori Yagiura, and Toshihide Ibaraki. An Iterated Local Search Algorithm for the Time-Dependent Vehicle Routing Problem with Time Windows. *Discrete Optimization*, 5:434–456, 2008.

[80] Moritz Hilger. Accelerating Point-to-Point Shortest Path Computations in Large Scale Networks. Master's thesis, Technische Universität Berlin, 2007.

[81] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In Demetrescu et al. [48].

[82] Martin Holzer, Frank Schulz, Dorothea Wagner, and Thomas Willhalm. Combining Speed-up Techniques for Shortest-Path Computations. *ACM Journal of Experimental Algorithmics*, 10(2.5):1–18, 2006.

[83] Martin Holzer, Frank Schulz, and Thomas Willhalm. Combining Speed-up Techniques for Shortest-Path Computations. In *Proceedings of the 3rd Workshop on Experimental Algorithms (WEA'04)*, volume 3059 of *Lecture Notes in Computer Science*, pages 269–284. Springer, 2004.

[84] Soumia Ichoua, Michel Gendreau, and Jean-Yves Potvin. Vehicle Dispatching with Time-Dependent Travel Times. *European Journal of Operational Research*, 144:379–396, 2003.

[85] Soojung Jung and Ali Haghani. Genetic Algorithm for the Time-Dependent Vehicle Routing Problem . *Journal of the Transportation Research Board*, 1771:164–171, 2001.

[86] Richard M. Karp. On-Line Algorithms Versus Off-Line Algorithms: How Much is it Worth to Know the Future? In *Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture - Information Processing '92*, pages 416–429. ACM Press, 1992.

[87] Richard M. Karp and James B. Orlin. Parameter Shortest Path Algorithms with an Application to Cyclic Staffing. *Discrete Applied Mathematics*, 2:37–45, 1980.

[88] Hanno Kersting. Algorithm Engineering in der Praxis am Fallbeispiel eines VRP. Master's thesis, Karlsruhe Institute of Technology, 2010.

[89] Tim Kieritz. Distributed Parallel Time Dependent Contraction Hierarchies. Master's thesis, Karlsruhe Institute of Technology, 2009.

[90] Tim Kieritz, Dennis Luxen, Peter Sanders, and Christian Vetter. Distributed Time-Dependent Contraction Hierarchies. In Festa [56].

[91] Emil Klafszky. Determination of shortest path in a network with time-dependent edge-lengths. *Mathematische Operationsforschung Statistik*, 3(4):255–257, 1972.

[92] Sebastian Knopp. Efficient Computation of Many-to-Many Shortest Paths. Master's thesis, Universität Karlsruhe (TH), Fakultät für Informatik, October 2006.

[93] Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. Fast Computation of Distance Tables using Highway Hierarchies. Technical report, Universität Karlsruhe (TH), Fakultät für Informatik, 2006.

[94] Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. Computing Many-to-Many Shortest Paths Using Highway Hierarchies. In ALENEX'07 [4], pages 36–45.

[95] Moritz Kobitzsch. Route Planning with Flexible Objective Functions. Master's thesis, Karlsruhe Institute of Technology, 2009.

[96] Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Acceleration of Shortest Path and Constrained Shortest Path Computation. In WEA'05 [140], pages 126–138.

[97] 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.

[98] Ulrich Lauther. An Experimental Evaluation of Point-To-Point Shortest Path Calculation on Roadnetworks with Precalculated Edge-Flags. In Demetrescu et al. [48].

[99] Jun Long. Developing high performance extensible transportation planning algorithm based on OSGI. Master's thesis, Karlsruhe Institute of Technology, 2009.

[100] Ronald Prescott Loui. Optimal Paths in Graphs with Stochastic or Multidimensional Weights. *Communications of the ACM*, 26(9):670–676, 1983.

[101] Chryssi Malandraki and Mark S. Daskin. Time Dependent Vehicle Routing Problems: Formulations, Properties and Heuristic Algorithms. *Transportation Science*, 26(3):185–200, 1992.

[102] Patrice Marcotte and Sang Nguyen, editors. *Equilibrium and Advanced Transportation Modelling*. Kluwer Academic Publishers Group, 1998.

[103] Ernesto Queiros Martins. On a Multicriteria Shortest Path Problem. *European Journal of Operational Research*, 26(3):236–245, 1984.

[104] Catherine C. McGeoch, editor. *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*. Springer, June 2008.

[105] Rolf H. Möhring. Verteilte Verbindungssuche im öffentlichen Personenverkehr – Graphentheoretische Modelle und Algorithmen. In Patrick Horster, editor, *Angewandte Mathematik insbesondere Informatik, Beispiele erfolgreicher Wege zwischen Mathematik und Informatik*, pages 192–220. Vieweg, 1999.

[106] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning Graphs to Speed Up Dijkstra's Algorithm. In WEA'05 [140], pages 189–202.

[107] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning Graphs to Speedup Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 11(2.8):1–29, 2006.

[108] Matthias Müller–Hannemann and Mathias Schnee. Finding All Attractive Train Connections by Multi-Criteria Pareto Search. In *Algorithmic Methods for Railway Optimization* [68], pages 246–263.

[109] Matthias Müller–Hannemann, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Timetable Information: Models and Algorithms. In *Algorithmic Methods for Railway Optimization* [68], pages 67–90.

[110] Matthias Müller–Hannemann and Karsten Weihe. Pareto Shortest Paths is Often Feasible in Practice. In *Proceedings of the 5th International Workshop on Algorithm Engineering (WAE'01)*, volume 2141 of *Lecture Notes in Computer Science*, pages 185–197. Springer, 2001.

[111] Karl Nachtigall. Time depending shortest-path problems with applications to railway networks. *European Journal of Operational Research*, 83(1):154–166, 1995.

[112] Giacomo Nannicini, Daniel Delling, Leo Liberti, and Dominik Schultes. Bidirectional A* Search for Time-Dependent Fast Paths. In McGeoch [104], pages 334–346.

[113] Sabine Neubauer. Space Efficient Approximation of Piecewise Linear Functions, 2009. Student Research Project. `http://algo2.iti.kit.edu/download/neubauer_sa.pdf`.

[114] Noam Nisan, Tim Roughgarden, Éva Tardos, and Vijay V. Vazirani, editors. *Algorithmic Game Theory*. Cambridge University Press, 2007.

[115] Masyuku Ohta, Kosuke Shinoda, Yoichiro Kumada, Hideyuki Nakashima, and Itsuki Noda. Is Dial-a-Ride Bus Reasonable in Large Scale Towns? — Evaluation of Usability of Dial-a-Ride Systems by Simulation —. In *Multiagent for Mass User Support - First International Workshop*, volume 3012 of *Lecture Notes in Computer Science*, pages 105–119. Springer, 2004.

[116] 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.

[117] Ariel Orda and Raphael Rom. Minimum Weight Paths in Time-Dependent Networks. *Networks*, 21:295–319, 1991.

[118] James B. Orlin, Neal E. Young, and Robert Tarjan. Faster Parametric Shortest Path and Minimum Balance Algorithms. *Networks*, 21(2):205–221, 1991.

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

[120] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Towards Realistic Modeling of Time-Table Information through the Time-Dependent Approach. In ATMOS'03 [6], pages 85–103.

[121] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Efficient Models for Timetable Information in Public Transportation Systems. *ACM Journal of Experimental Algorithmics*, 12(2.4):1–39, 2007.

[122] Michael Rice and Vassilis Tsotras. Graph Indexing of Road Networks for Shortest Path Queries with Label Restrictions. In *Proceedings of the 37th International Conference on Very Large Databases (VLDB 2011)*, 2011. To appear.

[123] Peter Sanders and Dominik Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05)*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2005.

[124] Peter Sanders and Dominik Schultes. Engineering Highway Hierarchies. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA'06)*, volume 4168 of *Lecture Notes in Computer Science*, pages 804–816. Springer, 2006.

[125] Peter Sanders and Dominik Schultes. Robust, Almost Constant Time Shortest-Path Queries in Road Networks. In Demetrescu et al. [48].

[126] Peter Sanders, Dominik Schultes, and Christian Vetter. Mobile Route Planning. In ESA'08 [55], pages 732–743.

[127] Dennis Schieferdecker. Systematic Combination of Speed-Up Techniques for exact Shortest-Path Queries. Master's thesis, Universität Karlsruhe (TH), Fakultät für Informatik, January 2008.

[128] Heiko Schilling. *Route Assignment Problems in Large Networks*. PhD thesis, Technische Universität Berlin, 2006.

[129] Dominik Schultes. *Route Planning in Road Networks*. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, February 2008. `http://algo2.iti.uka.de/schultes/hwy/schultes_diss.pdf`.

[130] Dominik Schultes and Peter Sanders. Dynamic Highway-Node Routing. In Demetrescu [47], pages 66–79.

[131] Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. In *Proceedings of the 3rd International Workshop on Algorithm Engineering (WAE'99)*, volume 1668 of *Lecture Notes in Computer Science*, pages 110–123. Springer, 1999.

[132] Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. *ACM Journal of Experimental Algorithmics*, 5(12):1–23, 2000.

[133] Robert E. Tarjan and Michael L. Fredman. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, July 1987.

[134] Dirk Theune. *Robuste und effiziente Methoden zur Lösung von Wegproblemen*. PhD thesis, Universität Paderborn, 1995.

[135] Mikkel Thorup. Integer Priority Queues with Decrease Key in Constant Time and the Single Source Shortest Paths Problem. In *Proceedings of the 35th Annual ACM Symposium on the Theory of Computing (STOC'03)*, pages 149–158, June 2003.

[136] Christian Vetter. Parallel Time-Dependent Contraction Hierarchies, 2009. Student Research Project. `http://algo2.iti.kit.edu/download/vetter_sa.pdf`.

[137] Christian Vetter. Fast and Exact Mobile Navigation with OpenStreetMap Data. Master's thesis, Karlsruhe Institute of Technology, 2010.

[138] Lars Volker. Route Planning in Road Networks with Turn Costs, 2008. Student Research Project. `http://algo2.iti.uni-karlsruhe.de/documents/routeplanning/volker_sa.pdf`.

[139] Dorothea Wagner and Thomas Willhalm. Geometric Speed-Up Techniques for Finding Shortest Paths in Large Sparse Graphs. In *Proceedings of the 11th Annual European Symposium on Algorithms (ESA'03)*, volume 2832 of *Lecture Notes in Computer Science*, pages 776–787. Springer, 2003.

[140] *Proceedings of the 4th Workshop on Experimental Algorithms (WEA'05)*, volume 3503 of *Lecture Notes in Computer Science*. Springer, 2005.

[141] Thomas Willhalm. *Engineering Shortest Paths and Layout Algorithms for Large Graphs*. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2005.

[142] Stephan Winter. Modeling Costs of Turns in Route Planning. *GeoInformatica*, 6(4):345–361, 2002.

[143] Yun Hui Wu, Lin Jie Guan, and Stephan Winter. Peer-to-Peer Shared Ride Systems. In *GeoSensor Networks*, volume 4540 of *Lecture Notes in Computer Science*, pages 252–270. Springer, August 2006.

[144] Xin Xing, Tobias Warden, Tom Nicolai, and Otthein Herzog. SMIZE: A spontaneous Ride-Sharing System for Individual Urban Transit. In *Proceedings of the 7th German Conference on Multiagent System Technologies (MATES 2009)*, volume 5774 of *Lecture Notes in Computer Science*, pages 165–176. Springer, September 2009.

Index

List of Notation

Zusammenfassung

Routenplanung umfasst eine Vielzahl interessanter Probleme, die sich algorithmisch lösen lassen. Gewöhnlich wird dazu ein Verkehrsnetz durch einen gewichteten Graphen modelliert, in dem die optimale Lösung des Problems durch kürzeste Wege beschrieben wird. Typische Verkehrsnetze sind Straßennetze oder öffentliche Verkehrsnetze. Der klassische Algorithmus zur Berechnung von kürzesten Wegen ist der Algorithmus von Dijkstra. Doch dieser ist zu langsam für sehr große Graphen, wie beispielsweise ein kontinentales Straßennetz mit etwa 100 Millionen Kanten. Deswegen wurden in den letzten Jahren immer schnellere Algorithmen entwickelt. Während einem Vorberechnungsschritt sammeln diese Algorithmen zusätzliche Daten, mit deren Hilfe sich die Berechnung kürzester Wege zwischen beliebigen Start- und Zielknoten beschleunigen lassen. Oftmals wird von diesen Algorithmen jedoch nur ein sehr einfaches Kürzeste-Wege-Problem betrachtet: den kürzesten Weg in einem Graphen mit reellwertigen, nicht-negativen Kantengewichten zwischen einem beliebigen Start- und einem beliebigen Zielknoten zu berechnen. Der verwendete Graph repräsentiert Straßenkreuzungen als Knoten und Straßen als Kanten. Das Gewicht einer Kante ist gewöhnlich die durchschnittliche Reisezeit auf der repräsentierten Straße oder die Länge der Straße. Doch dadurch lässt sich nur ein eingeschränktes Modell der Wirklichkeit erzeugen. Erweiterte praxisrelevante Probleme müssen weitere Eigenschaften berücksichtigen. Dazu gehören zeitabhängige Kantengewichte, Verbote für gewisse Fahrzeuge auf gewissen Straßen oder mehrere Start- und Zielknoten. Einige der existierenden schnellen Algorithmen können für einige dieser erweiterten Probleme zwar theoretisch eingesetzt werden, sind aber gewöhnlich nicht mehr effizient genug. Deswegen ist es wichtig, neue algorithmische Bauteile oder sogar vollkommen neue algorithmische Ideen zur Lösung dieser erweiterten Probleme zu entwickeln. Die dadurch neu entstandenen Algorithmen haben einen direkten Praxisbezug und werden auch schon teilweise in der Praxis eingesetzt. Der Fokus dieser Arbeit liegt auf den folgenden drei erweiterten Problemgebieten: *öffentliche Verkehrsnetze*, *flexible Anfragen* und *Serienberechnung kürzester Wege*. Im Folgenden wird jedes dieser Problemgebiete genauer beschrieben:

Öffentliche Verkehrsnetze, wie beispielsweise Bahn- und Busnetze, sind inhärent ereignisbasiert. Deren zeitabhängige Modellierung ist deswegen notwendig. Auch gibt es darin eine geringere hierarchische Struktur als in Straßennetzen, was die Effizienz von schnellen Algorithmen für Straßennetze deutlich reduziert. In realistischeren Szenarien ist deren Einsatz sogar nicht praktikabel, da keine zufriedenstellenden Laufzeiten erzielt werden können. Diese Arbeit analysiert zum einen die mangelnde Effizienz und zeigt Lösungsmöglichkeiten auf. Zum anderen entwickelt sie einen neuen Algorithmus speziell für öffentliche Verkehrsnetze. Das Konzept der Knotenkontraktion ist sehr effizient für Straßennetze. Eine Anwendung auf Graphen von öffentlichen Verkehrsnetzen war bisher aufgrund der Modellierung einzelner Haltestellen durch mehrere Knoten ineffizient. Der Vorteil dieser Modellierung lag in vereinfachten Kantengewichten. Ein neu entwickeltes Modell, welches nur einen Knoten pro Haltestelle besitzt und parallele

Kanten vermeidet, steigert die Effizienz der Kontraktion. Weitere algorithmische Bauteile machen diese noch effizienter. Ein komplett neuer Algorithmus für einen völlig realistischen Routenplaner für öffentliche Verkehrsnetze basiert auf der Idee von Umsteigemustern: die Folge von Haltestellen an denen umgestiegen wird. Gewöhnlich reicht schon eine kleine Menge von Umsteigemustern aus, um alle optimalen Verbindungen zwischen zwei Haltestellen, unabhängig vom Abfahrtszeitpunkt, zu beschreiben. Somit stellt die Traversierung dieser Umsteigemuster einen effizienten Anfragealgorithmus dar. Das Hauptproblem ist die Berechnung der Umsteigemuster. Eine Berechnung aller optimalen Verbindungen zwischen allen Paaren von Haltestellen verbietet sich aus Laufzeitgründen. Deswegen wurden Heuristiken für eine praktikable Berechnung entwickelt. Der resultierende Algorithmus ist in der Lage, Anfragen zwischen zwei Positionen, nicht notwendigerweise Haltestellen, effizient zu beantworten. Selbst für strukturschwache Netzwerke mit Hunderttausenden von Haltestellen beträgt die Antwortzeit nur 50 ms.

Der Begriff *flexible Anfrage* bezieht sich auf die Berechnung eines kürzesten Weges zwischen einem Start- und einem Zielknoten unter Berücksichtigung weiterer Anfrageparameter. Mögliche Anfrageparameter können beispielsweise die zugrunde liegende Kantenfunktion wählen (z. B. schnellste oder kürzeste Route) oder Verbote berücksichtigen (z. B. Fahrzeuge mit wassergefährdender Ladung). Obwohl der Algorithmus von Dijkstra sich leicht an solche Anfrageparameter anpassen lässt, trifft das im Allgemeinen nicht für schnellere, vorberechnungsbasierte Algorithmen zu. Theoretisch könnte die Vorberechnung separat für jeden möglichen Wert durchgeführt werden, den der Anfrageparameter annehmen kann. Praktisch ist dies jedoch äußerst ineffizient. Deshalb wurden Algorithmen entworfen, welche eine gemeinsame Vorberechnung durchführen. Gewöhnlich ist es einfach, kleine Verbesserungen gegenüber dem theoretischen Ansatz zu erzielen. Die Herausforderung bestand darin, Algorithmen zu entwerfen, die nicht viel schlechter als die schnellen unflexiblen Algorithmen sind. Des Weiteren speichert eine gemeinsame Vorberechnung alle Daten, um Anfragen mit beliebig gewählten Anfrageparametern zu beantworten. Für die Beantwortung einer einzelnen Anfrage mit festem Wert der Anfrageparameter ist jedoch im Allgemeinen nur ein Teil dieser Daten notwendig. Dies lässt sich ausnutzen, um die Antwortzeit einer Anfrage zu reduzieren. Ein weiteres Problem stellt die Klassifikation von Straßen und Straßenkreuzungen dar. Autobahnen sind beispielsweise sehr wichtig für die Berechnung schnellster Routen, aber weniger wichtig für kürzeste. Die sogenannte „Hierarchie“ des Netzes ändert sich mit dem Wert der Anfrageparameter. Dadurch wird die Effizient der vorherigen Algorithmen deutlich beeinflusst und es wurden neue Techniken zur Kompensation entwickelt. Die schnellsten entwickelten Algorithmen sind eine Kombination aus hierarchischen und zielgerichteten Techniken, welche Anfragen innerhalb von Millisekunden beantworten können.

Bei der *Serienberechnung kürzester Wege* werden viele Start- und Zielknoten gleichzeitig betrachtet. Das klassische Problem dazu ist die Berechnung einer Distanztabelle zwischen einer Startknotenmenge S und einer Zielknotenmenge T. Theoretisch kann jeder der vorherigen Algorithmen solch eine Tabelle berechnen. Viel schnellere Algo-

rithmen sind aber möglich, indem man ausnutzt, dass die kürzesten Wege Distanzen zwischen allen Paaren in $S \times T$ berechnet werden müssen. Distanztabellen sind beispielsweise wichtig bei der Lösung von Logistikproblemen und im Lieferkettenmanagement. Diese Arbeit beschreibt einen effizienten Algorithmus zur Berechnung zeitabhängiger Tabellen. Die Distanz, in diesem Falle die Reisezeit, ist dabei eine Funktion über die Abfahrtszeit, genauso wie die Kantengewichte des Graphen. Da ein Feld der Tabelle nun eine Funktion statt eine einzelne Zahl repräsentieren muss, spielt der Speicherverbrauch eine große Rolle. Durch die Approximation dieser Funktionen lässt sich der Speicherverbrauch um Größenordnungen reduzieren. Es existiert ein Kompromiss zwischen dem erlaubten Fehler und dem Speicherverbrauch. Auch sind die Basisoperationen zur Berechnung kürzester Wege auf Reisezeitfunktionen deutlich komplexer und langsamer als die entsprechenden Operationen für reellwertige Kantengewichte. Durch zusätzliche Berechnungsschritte gelingt es, die Anzahl dieser langsamen Operationen zu reduzieren. Außerdem gibt es weitere praxisrelevante Probleme, die eine Serienberechnung kürzester Wege erfordern, sich aber vom Problem der Distanztabellenberechnung unterscheiden. Durch Analyse der Problemstruktur lassen sich dafür effiziente Algorithmen konstruieren. Ein Algorithmus zur Vermittlung von Fahrgemeinschaften ist beispielsweise in der Lage, das Angebot mit dem kleinsten Umweg zu finden. Eine Anfrage auf einer Datenbank von tausenden von Angeboten kann damit innerhalb von Millisekunden beantwortet werden. Ein weiteres Beispiel ist die schnelle Berechnung der nächstgelegenen Sonderziele. Selbst für kleine Suchradien sind deutliche Verbesserungen gegenüber dem Algorithmus von Dijkstra möglich.