Algorithm Engineering for Realistic Journey Planning in Transportation Networks

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

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

genehmigte

Dissertation

von

Thomas Pajor

aus Potsdam

Tag der mündlichen Prüfung:	15. November 2013
Erste Gutachterin:	Prof. Dr. Dorothea Wagner
Zweiter Gutachter:	Prof. Dr. Matthias Müller-Hannemann

Acknowledgements

First of all, I want to thank my advisor Dorothea Wagner for encouraging me to join her group. During the whole time she supported me in every possible way, and her advise was always inspiring and helpful. The warm, respectful and friendly atmosphere she created made it a very pleasant environment to work in. Above all, however, I am truly grateful that, without hesitation, she encouraged and supported me to spend two summers as a research intern at Microsoft Research in Mountain View, California and two months with Microsoft Consulting Services in Reading, England.

For that, I want to thank Daniel Delling, Andrew Goldberg, Roy Levin, and Renato Werneck, who invited me to Microsoft Research, and Hatay Tuna, Kutay Tuna, and Simon Williams, who asked me to join Microsoft Consulting Services; I am humbled to have been given these opportunities. I had two very productive summers in a great research lab and a fantastic time in England—both of which greatly contributed to this thesis.

Also, I want to thank Matthias Müller-Hannemann for willingly accepting the task of reviewing this thesis and the German Research Foundation (DFG) for financing my work at KIT.

Next, I want to give special thanks to my office mates in Karlsruhe and Mountain View: Andreas Gemsa with whom I always had a lot of fun, keeping us distracted when we needed it, and Ahswin Prasad and Ilya Razenshteyn for having two awesome summer internships. I would like to highlight my coauthors Moritz Baum, Julian Dibbelt, Bastian Katz, Dominik Kirchler, Leo Liberti, Martin Nöllenburg, Ignaz Rutter, Ben Strasser, Roberto Wolfler Calvo, Christos Zaroliagis, and Tobias Zündorf with whom I had many fruitful collaborations. I could always learn a lot from them. I also want to thank all my coworkers for a great time, especially Tanja Hartmann and Markus Völker for sharing countless coffee breaks chatting, and of course everybody who regularly dropped by our office after we moved downstairs to the second floor.

Furthermore, I would like to thank Lilian Beckert, Elke Sauer, Bernd Giesinger (from Karlsruhe), and Lori Blonn and Hugo Hernandez (from Mountain View), who were

magnificent at taking care of all the administrative things. I also want to thank everybody, who is not mentioned explicitly by name, but who directly or indirectly supported or contributed to this thesis.

Outside work, I would like to give heartfelt thanks to my friends for sharing great times together, my parents, who always supported me in my endeavors and without whom this thesis would not be possible, and especially my dear Benni for all the love and support over the years.

Abstract

Route planning in transportation networks is a fundamental problem with numerous interesting applications. Probably one of the best known applications are navigation devices for cars. More examples include map services on the Internet and timetable information systems, such as the one by Deutsche Bahn. In all cases, underlying algorithms must quickly compute optimal solutions for any query from the customer.

In this thesis, we introduce algorithmic solutions for the following topics: Journey planning in public transit and multimodal networks, customizable route planning in road networks, and the computation of jogging routes in pedestrian networks. The presented algorithms thereby exploit the structural properties of the underlying transportation networks explicitly. To obtain efficient and practical algorithms, we base our methodology of research on the paradigm of *Algorithm Engineering* [San09, MS10, SW13]. It is characterized by a cycle consisting of four steps: Algorithm design, theoretical analysis, implementation, and experimentation.

Public Transit Journey Planning. For the case of journey planning in public transit networks, the input is given as a *timetable*. It defines stops and trips (buses, trains, etc.), which operate along sequences of stops at certain times of the day. This thesis introduces a new multicore algorithm that computes *one-to-all range queries*. They ask for a set of optimal (regarding travel time) journeys all departing within a certain time range from one stop to *all* other stops of the network. The algorithm is based on a newly introduced graph model and carefully exploits the fact that journeys may dominate each other, which significantly reduces the search space size. The obtained query times are practical, even on dense metropolitan networks for a time range of a full day. This enables precomputation of a full distance table over a subset of important stops of the network. By these means, the very same algorithm can be further accelerated, if one is only interested in queries between pairs of stops.

Besides travel time, another—just as important—criterion is the number of transfers. To give the user a sensible set of alternative journeys, this thesis considers computing *Pareto sets* of nondominating journeys regarding travel time and the number of transfers. Here, state-of-the-art approaches use variants of Dijkstra's algorithm, which is slow in practice. In this work, a novel algorithm, which operates directly

on the underlying timetable, is presented. It, therefore, neither requires a graph, nor a priority queue. Instead, it exploits the fact that vehicles operate on well-defined routes, which allows for a dynamic program that successively constructs the Pareto set. The algorithm is very cache-efficient and faster by an order of magnitude than previous (graph-based) approaches. It answers queries in dense metropolitan networks within a few milliseconds. Moreover, we parallelize it and extended it to handle further criteria. Since it does not require preprocessing, it can be directly used in dynamic scenarios, easily handling delays and trip cancellations.

Multimodal Journey Planning. Another scenario considered in this thesis is multimodal journey planning. Here, one is interested in *integrated* algorithms, that combine different modes of transport in a reasonable way. A common approach to obtain feasible mode sequences is the *label constrained shortest path problem*, which models mode sequences by regular languages. A variant of Dijkstra's algorithm that runs on the union of each modal subnetwork computes provably optimal solutions, but is too slow in practice. This work presents a faster approach, which is based on the concept of vertex contraction. It preprocesses the input such that arbitrary mode sequences are retained. This enables the user to specify mode sequence contraints at query time, a problem considered challenging before.

Sometimes, the user is unwilling to (or simply cannot) state feasible mode sequences. Instead, it might be preferable to provide the user with a choice from a set of *concice* and *diverse* alternative journeys. Therefore, in this thesis, an approach is considered that combines multimodal and multicriteria route planning. Instead of obeying specific modal sequences, it identifies, for each mode of transportation, a *convenience criterion*. These criteria are then used to compute Pareto sets of alternative journeys. Here, one particular challenge is that the resulting sets may contain hundreds of insignificant solutions. They (unnecessarily) increase computation time and are of little value to the user. Therefore, based on fuzzy set theory, a *fuzzy dominance criterion* is used that is successful in extracting the k most relevant journeys.

Customizable Route Planning in Road Networks. In this part we revisit the classical problem of computing optimal routes in *road networks*. For most existing efficient algorithms, an update of the metric (e. g., because of a new traffic situation) requires rerunning a costly preprocessing phase. Our approach addresses this issue and is based on the (known) concept of multilevel overlay graphs. The key idea is to split the preprocessing phase: In a first (potentially slow) metric-independent stage, the graph is partitioned into loosely connected regions of roughly equal size. This defines the topology of the overlay graphs. The second metric-dependent stage then quickly computes weights on the arcs of the overlay graphs. Integrating a new metric only requires rerunning the second stage. This takes mere seconds in practice and enables new applications, such as real-time traffic or personalized cost functions.

Computation of Jogging Routes. The last part of this thesis considers computing "good" *jogging routes*: Given a source vertex in a pedestrian network and a length (of the desired route), it asks to compute a cycle containing the source vertex that approximates the given length. Moreover, an ideal route might have a rather circular shape and travel through nice areas (such as parks and forests) of the map. In this thesis, two approaches to solve this problem are presented. The first successively extends a (given) route by joining adjacent faces of the network. The second transfers the intuition of constructing equilateral polygons to graphs in order to obtain jogging routes. The algorithm can be easily parallelized and even computes sensible alternative routes.

Contents

Chapter 1

Introduction

Anybody who travels frequently knows the numerous journey planning services that are available today. In fact, such services have become quite ubiquitous and are more or less taken for granted. Examples include online services like Bing Maps, GPS navigation devices for private vehicles, or the journey planning systems offered by many public transit agencies, such as Deutsche Bahn [HaC]. Thereby, these services are usually either offered online on the Internet or offline by mobile devices and smartphone applications.

A key component of any journey planning service is an *algorithm* that computes the actual journeys for given pairs of source and destination locations. These algorithms must be fast, and they should provide exact and optimal solutions for any query requested by the customer. A common approach to this problem models the transportation network as a directed graph whose arc weights represent the metric (travel time, distance, etc.) one aims to optimize. Dijkstra's algorithm, which has been already introduced in the year 1959, can then be used to compute provably optimal journeys between vertices of this graph. Unfortunately, Dijkstra's algorithm is too slow on realistic inputs of country or continental scale to be practical: Answering a single query takes several seconds, even on current server hardware. Therefore over the last years, a plethora of research focused on accelerating Dijkstra's algorithm by utilizing an offline preprocessing phase. The fastest available techniques achieve—after a few minutes or hours of preprocessing—query times in the order of microseconds or less on networks of continental scale. This is up to seven orders of magnitudes faster than Dijkstra's algorithm.

However, most of these methods were developed with road networks and travel time metric in mind. While they can, in principle, be augmented to other types of networks (such as public transit), most methods lose their excellent performance on them. Moreover, besides computing only one (quickest) route, on these networks one is often interested in more complicated query scenarios. For example, in public transit networks it is often desirable to compute a *set* of optimal journeys that depart

within a specified *time range*. Also, only optimizing a single criterion, like travel time, may not be sufficient. In practice, considering further criteria, such as the number of transfers, is just as important.

The ultimate journey planner should go even further and consider different modes of transportation like car travel, walking, bicycles, public transit, and flights in a holistic approach. We call this scenario *multimodal* journey planning. Clearly, developing algorithms that compute optimal journeys in this scenario includes at least the challenges from each individual mode of transport. On top of that, different modes of transport must be combined in a *reasonable* way. For example, requiring the customer to use their private car between train rides may be infeasible. Also, some customers may prefer to use their bicycle for parts of their journey, while others may not. Any multimodal journey planning algorithm should explicitly consider such constraints and, ideally, provide concise and diverse sets of alternative journeys to the customer.

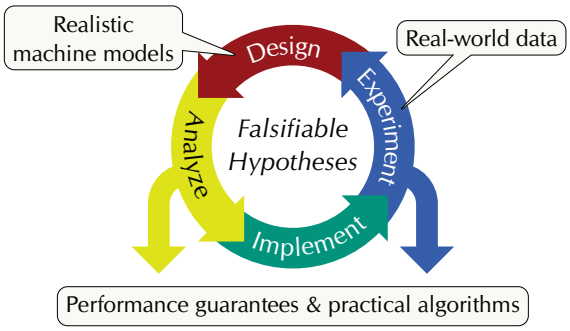


Figure 1.1. The *Algorithm Engineering* paradigm.

One of the main goals of our work is to develop algorithms that are both efficient and practical. Therefore, it is crucial to implement the algorithms carefully and conduct extensive experimental studies using real world data. The outcome of these experiments then gives new insights into the behavior of an algorithm, which in return, leads to a possibly refined design of the algorithm. This circular process is captured by the paradigm of *Algorithm Engineering* [San09, MS10, SW13]. At its core, it constitutes a cycle of algorithm design, theoretical analysis (in our case ensuring correctness), careful implementation, and extensive experiments, ideally, on real world data. The algorithm is thereby already designed with the underlying hardware and the characteristics of the inputs in mind. We then claim falsifiable hypotheses about its performance which are validated (or falsified) by the experiments. In that, Algorithm Engineering resembles Popper's scientific method [Pop34]. Also see Figure 1.1 for an illustration of the principle.

1.1. Main Contributions

This thesis contains contributions on the following topics: *Public transit journey planning*, *multimodal journey planning*, *customizable route planning in road networks*, and the *computation of jogging routes*. In this section we highlight the key contributions for each of these topics in turn.

1.1.1. Public Transit Journey Planning

For the problem of computing journeys in public transit networks, we consider *timetables* as input. Roughly speaking, a timetable is comprised of a set of stops (e. g., platforms, bus stops, etc.), a set of routes (e. g., bus lines), and a set of trips. A trip is thereby a vehicle that serves a route at a specific time of the day. Typically, the timetable is translated into a graph on which shortest paths correspond to optimal journeys. Several such graph models exist, incorporating different levels of realism.

Coloring Model and Footpaths. We present a new realistic time-dependent graph model, called the *Coloring Model*, which is useful to compute journeys that minimize arrival time. The model is based on the realistic time-dependent model from [PSWZ08]. By computing conflicting trips at stops in a principled way, we obtain significantly smaller graphs (up to a factor of 12). This immediately accelerates *any* query algorithm that runs on this model. Moreover, we present a new heuristic to generate artificial *footpaths* that connect stops which are close to the same intersection (of the underlying road network). Such footpaths are crucial to obtain realistic journeys, but are often missing from real world timetable data.

The Coloring Model is presented in Section 4.3.4 and our footpath heuristic is introduced in Section 4.3.5.

Parallel Range Query Algorithm. Based on our Coloring Model, we introduce a novel algorithm, called *Self-Pruning Connection-Setting Algorithm* (SPCS) that computes one-to-all range queries: For a given source stop p_s, such a query asks for optimal (regarding travel time) journeys departing within a given time range to *all* stops of the network. Unlike previous algorithms, it systematically exploits the combinatorial structure of public transit networks: The number of relevant connections to travel from p_s is limited and can be bounded in advance. In addition, certain connections are dominated by others along the way. Exploiting these principles in a sound manner, we augment the label-setting property of Dijkstra's algorithm to obtain a *connection-setting* algorithm for range queries in public transit networks. Unlike previous algorithms, which are notoriously hard to parallelize, SPCS admits a natural and efficient parallelization. As a result, SPCS is a more efficient substitute for Dijkstra's algorithm for the scenario of one-to-all range queries. Such queries are of particular importance as an ingredient to the preprocessing phase of many speedup techniques. Finally, for the case one is only interested in journeys to a designated target stop, we show how SPCS itself can be utilized for valuable preprocessing to further accelerate point-to-point queries.

Our new Self-Pruning Connection-Setting algorithm is presented in Section 4.5.

RAPTOR. Besides optimizing arrival time, another important criterion in public transit networks is the number of transfers. We, therefore, consider the problem

of computing Pareto sets of nondominating journeys regarding arrival time *and* number of transfers. To this extent, we present a novel algorithmic approach called *Round-bAsed Public Transit Optimized Router* (RAPTOR). Unlike previous algorithms, it is neither graph-based, nor does it require a priority queue. Instead, it organizes the timetable data efficiently into a small number of arrays. The algorithm then operates on these arrays in rounds (one per transfer) and scans each route of the timetable at most once per round. Essentially, the algorithm boils down to a dynamic program with simple data structures and excellent memory locality. By these means, query performance on the full metropolitan network of London is faster by an order of magnitude compared to previous algorithms. Moreover, we extend RAPTOR to handle strict dominance, multicriteria range queries, and additional criteria. In particular, we consider fare zones and reliability of transfers as additional criteria (besides arrival time and number of transfers) and present optimized variants of McRAPTOR (the more-criteria variant of RAPTOR) for them. Since RAPTOR does not rely on preprocessing, it can be directly used in dynamic scenarios, including delays, route changes, and trip cancellations.

Our new RAPTOR algorithm is presented in Section 4.6.

1.1.2. Multimodal Journey Planning

The second part of this thesis deals with *multimodal* journey planning. Here, we ask for a holistic algorithmic approach that computes journeys that *reasonably* combine different modes of transportations. In this work, we consider car travel, walking, rental bicycles, public transit, and flights as transportation modes.

User-Constrained Contraction Hierarchies. A quite elegant approach to the multimodal journey planning problem computes label-constrained shortest paths [BJM00]. Essentially, it imposes restrictions on the sequences of transportation modes in form of a regular language, to which any computed journey must obey. Although Dijkstra's algorithm can be augmented to handle such constraints, its performance is too slow to be practical. To this extent, we present a preprocessing-based speedup technique, called *User-Constrainted Contraction Hierarchies* (UCCH). It augments the Contraction Hierarchies algorithm [GSSV12] to handle label-constrained shortest paths in a sound manner. By ensuring that shortest paths for any mode sequences are retained during the preprocessing phase, we obtain the first preprocessing-based algorithm that can handle arbitrary mode sequence constraints as an input to the *query*—a problem considered challenging before. Moreover, when compared to previous algorithms with similar query performance (such as Access Node Routing [DPW09a]), UCCH has some key advantages: It does not require a dedicated algorithm to compute local queries, has faster preprocessing time, and can handle multimodal networks with a much denser public transit subnetwork.

The User-Constrained Contraction Hierarchies algorithm is presented in Section 5.3.

Multimodal Multicriteria Journey Planning. Even though label-constrained shortest paths can be used to forbid infeasible sequences of transportation modes (such as using a private car between two train rides), the customer still has to specify—and thus know—these constraints *in advance*. Preferably, a multimodal journey planner should provide the customer with a concise and diverse set of alternative journeys, from which they can choose their favorite option. To this extent, we drop label-constraints and consider to combine *multimodal* with *multicriteria* journey planning. We argue that users optimize—besides arrival time—specific mode-dependent convenience criteria. Examples include the number of transfers for public transit, walking duration for walking, and monetary cost for taxis. We present a new algorithm, called *multimodal multicriteria RAPTOR* (MCR), that builds on the round-based framework of RAPTOR and computes exact Pareto sets of journeys that optimize these convenience criteria. However, it turns out that these Pareto sets contain too many insignificant journeys with little value to the user. Therefore, we propose to use *fuzzy logic* to extract a subset of the most significant journeys in a quick postprocessing step. Going further, we present several heuristics (still multicriteria) that relax domination *during* the algorithm. They avoid computing insignificant journeys, but still closely match the best journeys of the exact Pareto set. Our experiments on the full multimodal network of London confirm that we are able to compute multimodal multicriteria journeys of high quality for large metropolitan areas.

Multimodal multicriteria journey planning and MCR are presented in Section 5.4.

1.1.3. Customizable Route Planning in Road Networks

The third part of this thesis considers the computation of shortest paths in road networks. While most research focused on fast methods that optimize travel time, we address the *customizable* route planning problem. Its goal is a method that is *metric-independent*: It must incorporate new metrics quickly, have only little space overhead (per metric), and admit a query algorithm that is *robust* with respect to *any* metric. To this extent, we analyze previous algorithms with regard to our scenario and propose an approach that is based on overlay graphs [SWW00, JP02]. To achieve our goals, we split the preprocessing phase into a *metric-independent preprocessing stage* and a *customization stage*. The first stage considers only topology, may take several minutes, and must be run only once. The customization stage then incorporates a new metric, which takes mere seconds, even for the continental network of Europe. Queries, which utilize the overlay graph, take a few milliseconds. This is fast enough for interactive scenarios. By these means, our approach is highly practical and enables new applications with obvious attraction: Traffic updates can be incorporated in real time, and customers may state personalized cost functions, such as “avoid highways”, “avoid toll roads”, “height restrictions for trucks”, and others. In fact, the proposed method is currently the core of the routing engine in use by Bing Maps [Mic12].

Our approach to customizable route planning is presented in Section 6.

1.1.4. Computation of Jogging Routes

The final part of this thesis considers the computation of *jogging routes* in pedestrian networks. To the best of our knowledge, we are the first to consider practical algorithms for this problem: Given a source vertex s and a desired length L, it asks for a simple cycle that contains s, whose length approximates L. Besides length, we identify further *soft criteria* one is usually interested to optimize: The route should have a rather circular shape, pass through "nice" areas of the map (such as parks and forests), and it should not contain too many turns that have to be remembered by the user. We show that the problem is NP-hard, even for the simple version that only considers length. Nevertheless, we present two novel and practical algorithms to compute sensible jogging routes heuristically. The first, called *Greedy Faces* (GF) iteratively extends the route by attaching adjacent faces of the graph. The second, called *Partial Shortest Paths* (PSP), concatenates several shortest paths with respect to an appropriate metric, and is based on the intuition of constructing equilateral polygons. The latter approach can be parallelized quite easily and inherently computes admissible alternative routes. We validate our algorithms in a systematic experimental study and present a case study on the map of Karlsruhe. The outcome of the experiments indicates that our algorithms are indeed able to compute sensible jogging routes fast enough for interactive applications.

The computation of jogging routes is considered in Chapter 7.

1.2. Outline

The rest of this thesis is organized as follows:

Chapter 2 gives an extensive overview on the current state-of-the-art that is related to this work. It recaps methods for route planning in road networks, journey planning in public transit networks, and journey planning in multimodal networks.

Chapter 3 settles (mathematical) notation that is fundamental to this work. In particular, it gives a formal introduction to graphs, shortest paths, partitions, and regular languages.

Chapter 4 contains the first main contribution of this thesis. It considers journey planning in public transit networks. In order to present our new algorithms, the chapter first gives a detailed introduction on the inputs, i. e., timetables (Section 4.1), the considered problems (Section 4.2), related graph models (Section 4.3), and basic algorithmic approaches (Section 4.4), which we use as baseline in our experiments. Going from there, the chapter introduces our new Self-Pruning Connection-Setting algorithm (Section 4.5) and RAPTOR (Section 4.6).

Chapter 5 contains the second main contribution of this thesis and considers journey planning in multimodal networks. We first analyze models for the individual modes of transport that make up our multimodal networks (Section 5.1) and present

journey planning problems that arise in the context of them (Section 5.2). In Section 5.3 we present User-Constrained Contraction Hierarchies, our new speedup technique that computes label-constrained journeys. Finally, our new MCR algorithm for multicriteria multimodal journey planning that uses fuzzy set theory to identify significant journeys, is presented in Section 5.4.

Chapter 6 contains the third main contribution of this thesis. It considers customizable route planning in road networks. We start by analyzing the shortcomings of existing algorithms with respect to our scenario (Section 6.1). We then present our approach to customizable route planning (Section 6.2) and show how it can be implemented efficiently (Section 6.3) on realistic road networks with turn costs (Section 6.4). Furthermore, we present detailed experiments (Section 6.5), describe how we retrieve the full path description (Section 6.6), and give some implementation details (Section 6.7). We conclude the chapter in Section 6.8.

Chapter 7 contains the last main contribution of this thesis. Section 7.1 formally defines the Jogging Problem and proves its NP-hardness. Section 7.2 presents our two novel algorithmic approaches: Greedy Faces and Partial Shortest Paths. Finally, Section 7.3 contains our experimental study while Section 7.4 summarizes the results and contains some interesting open questions.

Chapter 8 concludes our work with a summary of the most important results and discusses interesting open problems for future research.

Chapter 2

Literature Overview

This chapter gives an overview on state-of-the-art in algorithmic approaches for route planning that are related to this work. We start in Section 2.1 with route planning in road networks. Section 2.2 addresses work on journey planning in public transit networks, which have different properties from road networks. Finally, in Section 2.3 we present related work on multimodal journey planning that integrates road and public transit networks—among others—in a holistic approach.

To recap some of the techniques, we may use mathematical notation for graphs, partitions, and other things. A precise definition of these notions is given in Chapter 3.

2.1. Route Planning in Road Networks

Route planning in road networks has received tremendous amount of attention. A well-known approach to compute (optimal) routes models the road network as a directed graph $G = (V, A)$ with associated (usually nonnegative) arc costs $\ell\colon A \to \mathbb{Z}_{\geq 0}$. Thereby, vertices correspond to intersections (of the road network) and arcs represent street segments. The cost function ℓ can be any metric, however, most research focused on travel time. In the so-constructed graph, a *shortest path* between vertices s and t then corresponds to the *optimal route* with respect to the cost function ℓ.

While algorithms that compute shortest paths in graphs—such as Dijkstra's seminal algorithm [Dij59]—have been around for over sixty years, it has only been recently that computers became powerful enough to handle realistic and large-scale road networks, such as that of a whole continent. Paired with the observation that Dijkstra's algorithm is too slow for interactive applications (queries take seconds, even today), this motivated research on *speedup techniques* (for Dijkstra's algorithm) [SWW99]: Under the assumption that shortest path queries occur significantly more often than changes to the graph, one can use a (somewhat costly) *preprocessing phase* that computes auxiliary data which then helps to accelerate the *query algorithm*. When in

2005 large road networks were publicly released for the 9th DIMACS Implementation Challenge [DGJ09], research on speedup techniques culminated in a downright "horse race" about the fastest query algorithm for road networks.

In the following, we give an overview on the most important techniques for route planning in road networks. Besides individual publications there are also survey articles on the topic: In [WW07] and [DSSW09a] overviews on speedup techniques for shortest path queries are given. Extensions to time-dependent shortest paths are discussed in [DW09b]. A recent survey [Som12] also covers (besides practical algorithms) theoretical results, such as distance oracles. Another survey over some heuristic methods (which are not the focus of this thesis) is given in [FSR06]. Finally, Figure 2.12 (on Page 23) summarizes performance of those surveyed methods, for whom experimental data on the European road network is available.

2.1.1. Basic Algorithms

Here, we give an overview on basic algorithms for the shortest path problem including Dijkstra's algorithm [Dij59]. These algorithms do not employ a preprocessing phase.

Classical Algorithms. Probably the most well-known approach to compute shortest paths on a weighted graph with nonnegative arc cost is Dijkstra's algorithm introduced in the year 1959 [Dij59]. Given a source vertex $s \in V$, it computes distances $\text{dist}(s, u)$ to every vertex $u \in V$ of the graph.

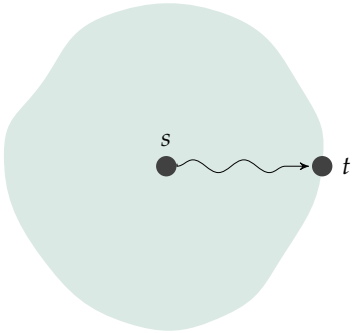


Figure 2.1. Schematic search space of Dijkstra's algorithm with stopping criterion.

Therefore, it maintains a priority queue Q of vertices ordered by their (tentative) distances from s. The algorithm initializes $\text{dist}(s, s) = 0$ and adds s to Q. In each iteration, it extracts (*scans*) the vertex u with minimum distance from Q and looks at all (to u incident) arcs $a = (u, v) \in A$. For each such arc, it determines the distance to v via arc a by computing $\text{dist}(s, u) + \ell(a)$. If this value improves $\text{dist}(s, v)$, the algorithm updates it and adds vertex v with key $\text{dist}(s, v)$ to the priority queue Q. Dijkstra's algorithm has the *label-setting* property, that is, once a vertex $u \in V$ has been extracted (scanned) from the priority queue, its distance value $\text{dist}(s, u)$ is correct and will not be improved anymore. Therefore, if one is interested in computing the distance to a dedicated target vertex t, the algorithm may stop as soon as it scanned t. The set of vertices $S \subseteq V$ scanned by the algorithm is called *search space* and consists of exactly those vertices $u \in V$ that have distance smaller than $\text{dist}(s, t)$. Note that S is actually a graph-theoretic disc centered at s with radius $\text{dist}(s, t)$. See Figure 2.1 for an illustration.

The running time of Dijkstra's algorithm is determined by the data structure that is chosen as priority queue Q. Using a binary heap, the running time is

in $\mathcal{O}((|V| + |A|) \log |V|)$ [CLRS01], which can be improved by, e. g., Fibonacci Heaps to $\mathcal{O}(|A| + |V| \log |V|)$ [FT87]. If all arc costs are integers in the range $[0, C]$, Multi-Level Bucket Queues yield a running time of $\mathcal{O}(|A| + |V| \sqrt{\log C})$ [DF79]. For sparse graphs (i. e., $|A| \in \mathcal{O}(|V|)$) the running time of Dijkstra's algorithm with binary heaps drops to $\mathcal{O}(|V| \log |V|)$. Note that better bounds exist for the average case [Mey01, Gol01].

If the cost function may assume negative values, but the graph does not contain negative cycles, simple shortest paths (i. e., paths that contain no vertex twice) can be computed by the Bellman-Ford algorithm [For56, Bel58] in time $\mathcal{O}(|V||A|)$. Another approach, based on Dijkstra's algorithm, may rescan vertices whenever a path with negative arcs improves its distance [DP84]. Moreover, the Floyd-Warshall algorithm [Flo62] computes distances between *all* pairs of vertices in time $\mathcal{O}(|V|^3)$ (requiring $\mathcal{O}(|V|^2)$ space). Note that, on sparse graphs with nonnegative arc weights, running $|V|$ times Dijkstra's algorithm yields a better running time of $\mathcal{O}(|V|^2 \log |V|)$.

Bidirectional Search. A first attempt to reduce the search space is *bidirectional search* [Dan62, GH05]. It simultaneously (and possibly in parallel) runs a backward search from the target vertex t. The algorithm may stop as soon as the intersection of the search spaces of the forward and backward search provably contains a vertex m on the shortest path from s to t. This is (roughly) the case when the searches meet. Also see Figure 2.2. While the theoretic running time does not improve that of Dijkstra's algorithm, in road networks the search space can be approximated by geometric discs: Bidirectional search grows two disks (centered at s and t) with radii $\frac{1}{2} \operatorname{dist}(s, t)$. Thus, if one considers the disc's area, the speedup over Dijkstra's algorithm is roughly

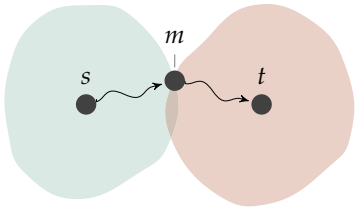


Figure 2.2. Schematic search space of bidirectional search.

$$\text{speedup} \approx \frac{\text{Dijkstra's algorithm}}{\text{bidirectional search}} = \frac{\pi \operatorname{dist}(s,t)^2}{2\pi \left(\frac{1}{2} \operatorname{dist}(s,t)\right)^2} = 2, \tag{2.1}$$

which is also observed in practice. While this seems limited, bidirectional search is nevertheless an important ingredient to many—especially hierarchical—methods.

2.1.2. Goal-Directed Techniques

Dijkstra's algorithm scans all vertices with distance smaller than $\operatorname{dist}(s, t)$. Goal-directed techniques, in contrast, aim to "guide" the search toward the target by avoiding to scan vertices that are not in direction of t. They either exploit the (geographical) embedding of the network or graph-theoretic properties, such as the structure of shortest path trees toward (connected) regions of the graph.

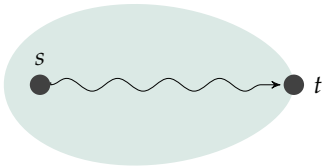


Figure 2.3. Schematic search space of the A* algorithm.

A* Search. A classic goal-directed shortest path algorithm is A* search [HNR68]. It uses a potential function $\pi\colon V \to \mathbb{R}$ on the vertices, which estimates the distance $\text{dist}(u,t)$ from u to t by a *lower bound*. It then essentially runs Dijkstra's algorithm, however, with the modification that it sets the key of a vertex u in the priority queue to $\text{dist}(s,u) + \pi(u)$. By these means the order in which vertices are scanned is altered such that vertices that are closer to the target t are scanned earlier during the execution of the algorithm. See also Figure 2.3. Note that if π were an *exact* lower bound, i. e., $\pi(u) = \text{dist}(u,t)$, *only* vertices along shortest s–t paths would be scanned. Precomputing exact potentials is, however, too expensive in practice. Therefore, in road networks with travel time metric (and coordinates associated with the vertices), one often uses the direct geographical distance [Poh71, SV86] between u and t divided be the maximum travel speed (that occurs in the network) as lower bound. Unfortunately, these bounds are poor, and the reduction in search space does not even weigh out the additional overhead of computing potentials in the algorithm [GH05].

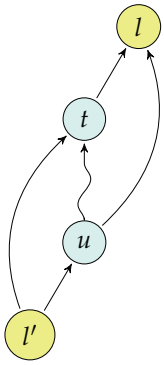


Figure 2.4. Triangle inequalities for ALT.

ALT. To obtain significantly better lower bounds, the *ALT* algorithm [GH05], which stands for *A*, landmarks, and triangle inequality*, computes, in a preprocessing phase, for a designated set of landmark vertices $L \subseteq V$ exact distances to and from all vertices in the graph. It then uses, for a selected landmark $l \in L$, these distances and the following triangle inequalities to obtain lower bounds on $\text{dist}(u,t)$ in the algorithm:

$$\begin{aligned} \text{dist}(u,t) + \text{dist}(t,l) \geq \text{dist}(u,l) \quad &\Rightarrow \quad \text{dist}(u,t) \geq \text{dist}(u,l) - \text{dist}(t,l), \\ \text{dist}(l,u) + \text{dist}(u,t) \geq \text{dist}(l,t) \quad &\Rightarrow \quad \text{dist}(u,t) \geq \text{dist}(l,t) - \text{dist}(l,u). \end{aligned}$$

Also see Figure 2.4 for an illustration of the inequalities. Note that landmark l is used to illustrate the first inequality, while l' illustrates the second.

Different landmark selection strategies exist, from which it turned out that selecting landmarks that are at the "far boundary" of the graph results in the best query performance on road networks [GW05]. Moreover, since lower bounds obtained from the above triangle inequalities are still correct if arc weights increase, ALT is robust with respect to dynamic scenarios that consider traffic data [DW07].

Geometric Containers. Another method to guide the search toward t, called *Geometric Containers*, precomputes, for each arc $a = (u,v) \in A$, an arc label $L(a)$ that encodes (at least) the set V_a of vertices to which a shortest path from u begins with the arc a. Instead of storing V_a explicitly, $L(a)$ approximates this set by using geometric information (i. e., the coordinates) of the vertices in V_a. Then, if during query the target vertex t is not contained in $L(a)$, it is also not contained in V_a, and the search

can be pruned at a. In [SWW00] the set V_a is approximated by an angular sector (centered at u) that covers all vertices in V_a. In [WWZ05] more complicated geometric containers such as rectangles, ellipses, and the convex hull are evaluated. From these, the bounding box consistently performs well. For the case that the graph is given without geometric information, in [BSWW01, WW05] several graph layout algorithms are evaluated with respect to the query performance of geometric containers.

Arc-Flags. A disadvantage of Geometric Containers is that its preprocessing essentially requires an all-pairs shortest path computation, which is costly.

Arc Flags [Lau97, Lau04, KMS05, HKMS09] uses a similar (to Geometric Containers) approach, but drops geometry. Instead, the graph is partitioned into K (balanced) regions with a preferably low number of boundary vertices. Each arc maintains a vector of K bits (arc flags), indicating toward which regions the arc a lies on a shortest path. The search algorithm then prunes arcs which do not have the bit set for the region which contains t. Figure 2.5 illustrates the method (example taken from [Del09]). Vertices and arc flags are colored with respect to their region. To further improve the query performance, Arc Flags can be extended to nested multi-level partitions [MSS+06]: Whenever the search reaches the region that contains t, it descends one level of the partition, i. e., it evaluates arc flags with respect to the (finer) cells of the next-lower level of the partition.

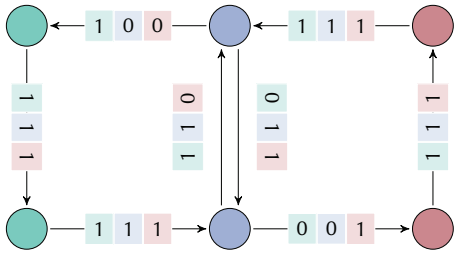


Figure 2.5. Arc flags for a small graph.

Arc flags for a region i are computed by growing a backward shortest path tree from each boundary vertex (of region i) and, thereby, setting the respective flag for all arcs of the tree. Alternatively, one can compute arc flags by running a label-correcting algorithm from all boundary vertices simultaneously [HKMS09]. Moreover, to reduce preprocessing space, one can use a (still correct) compression scheme which may flip flags from zero to one [BDGW10]. Arc Flags is currently the fastest (regarding query time) purely goal-directed method, with speedups of more than 5 000 over Dijkstra's algorithm on continental road networks [BDS+10]. Though high preprocessing times (of several hours) have long been a drawback of Arc Flags, the recent PHAST algorithm (mentioned later) computes arc flags within a few minutes [DGNW13].

Precomputed Cluster Distances. Another goal-directed technique is *Precomputed Cluster Distances* [MSM09]. Like Arc Flags, it is based on a (preferably balanced) partition $\mathcal{C} = (C_1, \ldots, C_K)$ with K cells (or clusters). During preprocessing, it computes, for each pair C_i, C_j of cells the shortest path distance between these cells, i. e.,

$$\operatorname{dist}(C_i, C_j) = \min_{\substack{u \in C_i \\ v \in C_j}} [\operatorname{dist}(u, v)]. \tag{2.2}$$

The query algorithm then maintains and minimizes a global upper bound μ on the length of the shortest s–t path by evaluating the precomputed cluster distances at vertices $u \in C_i$ which have been responsible for setting $\text{dist}(C_i, C(t))$. Moreover, at any vertex u, the algorithm obtains a lower bound on the shortest s–t path via u by evaluating $\text{dist}(s,u) + \text{dist}(C(u), C(t)) + \text{dist}(v,t)$, where v is the boundary vertex of cell $C(t)$ with minimal distance to t. The algorithm prunes vertices at which this lower bound exceeds μ. Query performance of PCD is similar to ALT, but it requires less preprocessing space.

2.1.3. Hierarchical Techniques

Hierarchical methods aim to exploit the inherent hierarchy of road networks (with travel time metric): Sufficiently long shortest paths eventually converge to a small arterial network of important roads, such as highways. Intuitively, once one is far from the source and target, it suffices to only scan vertices of this subnetwork in the algorithm. The following methods formalize this notion.

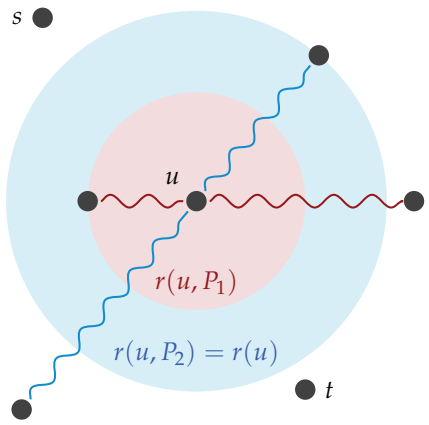


Figure 2.6. Reach of vertex u determined by the paths P_1 (red) and P_2 (blue). An s–t query may prune u.

Reach. The first algorithm that formalizes this observation is *Reach* [Gut04]. Reach is a centrality measure defined on the vertices: Let P be a shortest s–t path that contains vertex u. Then, the reach $r(u,P)$ of u on P is defined as $\min(\text{dist}(s,u), \text{dist}(u,t))$. Based on this, the (global) reach of u in the graph G is the maximum reach of u over *all* shortest paths that contain u. Now, for given reach values the query algorithm prunes the search at any vertex u, for which both $\text{dist}(s,u) > r(u)$ *and* $\text{dist}(u,t) > r(u)$ hold true: The shortest path from s to t does provably not contain u. To check these conditions, the algorithm runs a bidirectional search (cf. Section 2.1.1) from s and t and extracts lower bounds on $\text{dist}(u,t)$ (forward search) and $\text{dist}(s,u)$ (backward search) from the respective opposite search direction [GKW09]. Also see Figure 2.6 for an illustration of reach.

Determining exact reach values requires computing shortest paths for all pairs of vertices, which is too expensive on large road networks. However, the query is still correct if $r(u)$ only depicts an upper bound on the reach of u. Such bounds can be obtained faster by computing partial shortest path trees [Gut04] and by (additionally) adding shortcuts to the graph [GKW09].

Contraction Hierarchies. Another approach that exploits the hierarchy is based on the concept of *shortcuts* [SWW00]. A shortcut is an arc (u,v)—possibly not contained in the original graph—that represents a shortest path from u to v in G. The goal is

to augment G with shortcuts such that long-distance queries use these shortcuts in order to skip over "unimportant" vertices. *Contraction Hierarchies* [GSSD08, GSSV12] implements this idea by repeatedly executing an operation called *vertex contraction*.

During preprocessing, Contraction Hierarchies (heuristically) orders the vertices by an importance value and then contracts them in this order (from least to most important). To contract a vertex v, it is (temporarily) removed from G and shortcuts are created between each pair of neighboring vertices u, w, if the shortest path from u to w is unique and contains v. The query algorithm then runs a bidirectional search from s and t on G augmented by the shortcuts computed during preprocessing. Thereby, it only considers arcs to (forward search), respective from (backward search) vertices with higher ranks (regarding the contraction order). Also see Figure 2.7. Vertex orders are usually determined online and bottom-up. The algorithm selects the vertex to be contracted next, which minimizes a linear combination of degree, arc expansion, number of contracted neighbors, and other factors [GSSV12, KLSV10]. Better vertex orders can be obtained by combining the bottom-up algorithm with a (more expensive) top-down offline algorithm that is based on computing shortest path covers [ADGW12].

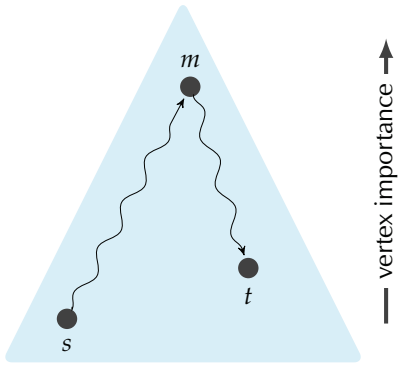


Figure 2.7. Illustrating a Contraction Hierarchies query.

Contraction Hierarchies turned out to be versatile and many extensions of the algorithm exist. Examples include time-dependent scenarios [BGSV13], dynamic scenarios [GSSV12], distributed preprocessing [KLSV10], external memory implementations [SSV08], road networks with turn costs [GV11, DGPW11], computing route corridors [DKLW12], obtaining alternative routes [ADGW13, LS12], ride sharing scenarios [GLS+10], minimizing energy consumption of electric vehicles [EFS11], handling flexible arc restrictions [GRST12], or handling multiple criteria [GKS10, FS13]. Moreover, Contraction Hierarchies can be extended to compute distances to all [DGNW13] or a restricted subset [KSS+07, DGW11] of the vertices. The algorithm is also used in practice, for example, on OpenStreetMap [Ope04] data of planetary scale [LV11]. Note that Contraction Hierarchies is a successor of Highway Hierarchies [SS05, SS12a] (and Highway Node Routing [SS07]), which are based on similar ideas.

2.1.4. Separator-Based Techniques

Though road networks are not necessarily planar (think of bridges or tunnels), it has been observed that they nevertheless have small separators [EG08, DGRW11, SS12b]. This fact is exploited by the methods in this section. Note that separator-based algorithms may also classify as hierarchical techniques (cf. Section 2.1.3).

Vertex Separators. The first class of algorithms is based on *vertex separators*: A vertex separator is a (preferably small) subset $S \subset V$ of the vertices, such that removing S from V decomposes the graph G into several (preferably balanced) cells (components). This separator is then often used to compute an *overlay graph* over S: Shortcuts are added to the overlay such that distances between *any* pair of vertices from S are preserved, i. e., they are equivalent to the distance in G. A query algorithm may then use the much smaller overlay graph for (parts of) the query.

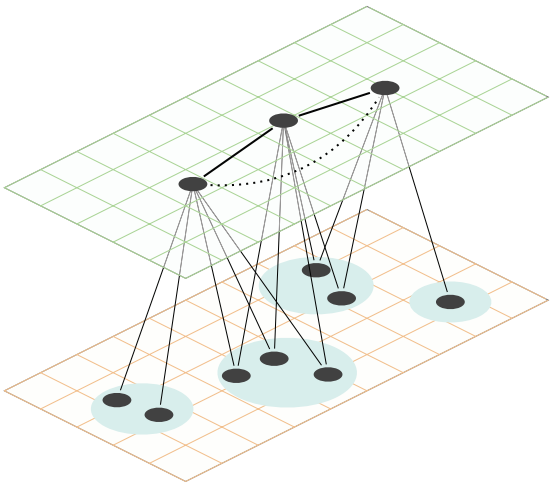


Figure 2.8. Multilevel overlay graph with two levels. The dots depict separator vertices in the lower (orange) and upper (green) level.

In [SWW00] an overlay graph over a carefully chosen subset S (not necessarily a separator) of "important" vertices is used: For each pair of vertices $u, v \in S$ an arc (u, v) is added to the overlay, if the shortest path from u to v in G does not contain any other vertex w from S. This approach is further extended in [SWZ02] to multilevel separator hierarchies $V \supset S_1 \supset S_2 \supset \cdots \supset S_k$ of k levels. In addition to arcs between separator vertices of the same level, the overlay contains, for each cell on level i, arcs between the confining level i separator vertices and the interior level $i-1$ separator vertices. See Figure 2.8 for an illustration. In [SWW00] and [SWZ02] performance is experimentally evaluated on time-expanded graphs from railway networks (cf. Sections 2.2 and 4.3.2). A systematic evaluation of the algorithm is available in [HSW08]: Besides testing separators obtained by different methods, such as by the Planar Separator Theorem [LT79] and METIS [Kar07], it also includes experiments on road networks (reporting speedups of above 50).

A highly engineered variant of the multilevel overlay graph approach is called *High-Performance Multilevel Routing* [DHM+09]. It achieves query times of 40 µs on the road network of Europe by adding many more shortcuts to the overlay in a first step and sparsing them out significantly in a second step. However, space consumption of the auxiliary data is very high and preprocessing times are in the order of a full day.

Arc Separators. The second class of algorithms uses *arc separators* (instead of vertex separators) to build the overlay graphs. Therefore, they compute in a first step a partition $\mathcal{C} = (C_1, \ldots, C_k)$ of the vertices, such that cells are balanced and the number of cut arcs is minimized. (Cut arcs connect boundary vertices of different cells.) The *Hierarchical MulTi* (HiTi) method [JP02] builds an overlay graph as follows: It is initially composed of all boundary vertices of the partition plus all cut arcs. Next,

for each cell C_i, and between each pair of its boundary vertices u, v, preprocessing adds a shortcut (u, v) to the overlay that represents the restricted (to cell C_i) shortest path from u to v in G. Thus, the overlay consists of $|\mathcal{C}|$ cliques, interconnected by cut arcs. See also Figure 2.9. The query algorithm then (implicitly) runs Dijkstra on the subgraph induced by the cells containing s and t plus the overlay. This approach can be extended beyond one level by using nested multilevel partitions. Unfortunately HiTi has only been tested on grid graphs [JP02].

A recent algorithm, called *Customizable Route Planning* (CRP) [DGPW11,DGPW14] (also see Chapter 6), builds on a similar approach, but is specifically engineered to meet the requirements of a real world (production) system. It handles turn costs, continuous maneuver restrictions, and arbitrary metrics. To achieve these goals, preprocessing is split into two phases: metric-independent preprocessing and customization. The first computes, besides the partition (for which it uses PUNCH [DGRW11]), the topology of the overlays. It does not represent them as graphs, but stores them as matrices in contiguous memory. The customization phase then computes the cost of the clique arcs quickly, bottom-up, and in parallel. Incorporating a new metric on the European road network takes mere seconds and consumes only few tens of Megabytes of space. Customization time can be reduced even further to a few hundred milliseconds by using alternate shortest path algorithms (such as Bellman-Ford) paired with (metric-independent) contraction [DW13]. Note that customization times are fast enough to enable real-time traffic updates and personalized cost functions.

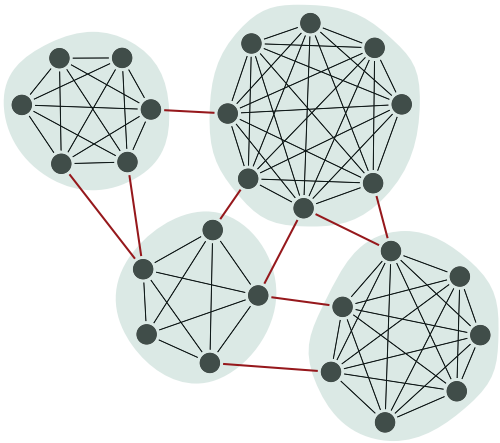


Figure 2.9. Overlay graph constructed from arc separators. Each cell contains a full clique between its boundary vertices, and cut arcs are drawn red.

The query algorithm runs (similarly to HiTi) a bidirectional search in the overlay graph. It takes time in the order of milliseconds, including full path retrieval [DGPW11]. This makes CRP very suitable for production systems: It has a practical tradeoff regarding query time, customization time, and space consumption. In fact, it is currently the core of the routing engine in use by Bing Maps [Mic12]. Moreover, CRP can be used to compute alternative routes [DGPW14] and has recently been extended to compute energy-optimal routes for electric vehicles [BDPW13].

Distance Oracles. Besides work on separator based methods from an algorithm engineering point of view, theoretic work on quick shortest path computation often also uses separator-based approaches. In particular, planar graphs have small separators of size $\mathcal{O}(\sqrt{|V|})$ [LT79]. Road networks are observed to also have small

separators [EG08], thus, theoretical techniques developed for planar graphs are likely to also perform well on road networks. A recent technique uses simple cycle separators [Mil86, FEMPS13] to construct, for any given parameter $S \in [|V| \log\log|V|, |V|^2]$, auxiliary data of size $\mathcal{O}(S)$ in time $\tilde{\mathcal{O}}(S)$. Then, queries can be answered in time $\tilde{\mathcal{O}}(|V|/\sqrt{S})$. Many other (also approximate) methods with different tradeoffs exist. Because the focus of this work is on algorithm engineering, we refrain from going into more detail about the available theoretic work. Instead, we refer the interested reader to the following overview articles: Sommer gives an excellent overview on many (not only theoretical) algorithms in [Som12]. Exact and approximate distance oracles are surveyed in [Zwi01] and [Sen09] and a survey with focus on route planning is given in [GP03].

2.1.5. Table-Based Techniques

Table-based methods precompute distances between important vertices such that all shortest path information is fully encoded by these tables. The query then only performs table lookups to retrieve distances. Algorithms in this category are not Dijkstra-based, i. e., no graph is explored during the query.

A naïve approach precomputes the distance for *all* pairs of vertices $u, v \in V$. A single lookup then suffices to retrieve the shortest distance. While (pre)computing all-pairs shortest paths has recently become feasible with the availability of the PHAST algorithm [DGNW13], space consumption of such a table (of size $|V|^2$) is prohibitive for realistic road networks.

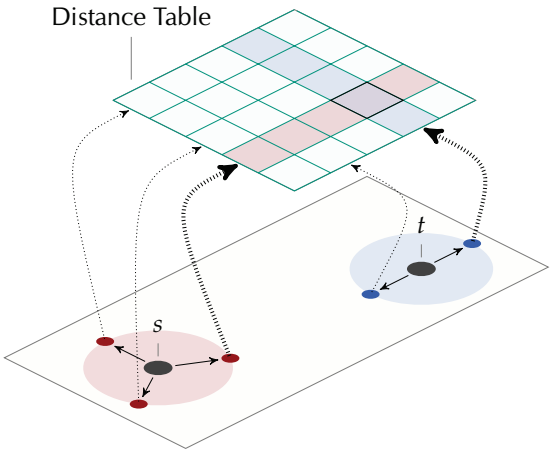


Figure 2.10. Illustrating a TNR query.

Transit Node Routing. A technique that uses distance tables on a subset of the vertices is called *Transit Node Routing* (TNR) [BFM+07, BFSS07, BFM09, SS09]. During preprocessing, it selects a small set $T \subseteq V$ of transit nodes and computes all pairwise distances between them. Moreover, it computes, for each remaining vertex $u \in V \setminus T$, its relevant set of transit nodes $A(u) \subseteq T$, called access nodes (of u). Access nodes are defined as follows: A transit node $v \in T$ is an access node of u, if there is a shortest path P from u in G such that v is the first transit node vertex contained in P. In addition to the vertex itself, preprocessing also stores the distances between u and its access nodes. Now, the query algorithm uses the distance table to select the path that minimizes the combined s–$A(s)$–$A(t)$–t distance. Note that the result is incorrect, if the shortest path does not contain a vertex from T.

Therefore, a *locality filter* first decides whether the query might be local (i. e., does not contain a vertex from T). In that case, a fallback shortest path algorithm is run to compute the correct distance. See Figure 2.10 for an illustration of a TNR query. The red (blue) dots are the access nodes of s (t). The arrows point to the respective rows/columns of the distance table. The highlighted entries correspond to the access nodes which minimize the combined s–t distance. Note that, for simplicity, we explained the algorithm for a single distance table. To obtain better performance, it is usually extended to multiple (hierarchical) levels of transit (and access) nodes [BFM09, SS09].

Crucial to the performance of the algorithm is the choice of the transit node set. Besides selecting vertex separators or boundary vertices of arc separators as transit nodes [Mül06, DHM+09, BFM09], vertices that are characterized as important by a hierarchical speedup technique (such as Contraction Hierarchies) [SS09, BFM+07, GSSV12, ALS13] work very well. The former approach admits a natural locality filter, while for the latter an efficient locality filter can be constructed by using the (graph-theoretic) Voronoi regions [Vor08, AKL13] that are induced by the transit nodes [ALS13].

Labeling Algorithms. Another framework of algorithms that reorganizes the shortest path structure of the network in order to perform distance queries is called *Labeling Algorithms* [Pel00]: During preprocessing, a *label* $L(u)$ is computed for each vertex u of the graph, such that, for any pair of vertices u, v, their distance $\text{dist}(u, v)$ can be determined by only looking at the labels $L(u)$ and $L(v)$. Interestingly, general graphs have labels of size at least $\Theta(|V|)$ [GPPR04], which is too large to be practical.

However, for networks with small highway dimension h [AFGW10], the following labeling algorithm admits labels of size $\mathcal{O}(\Delta h \log D)$. Here, Δ is the maximum degree and D the diameter of the graph. Note that in [AFGW10] road networks are conjectured to have small highway dimension. The label $L(u)$ of each vertex u consists of a set of vertices and their distances from u, such that the following cover property holds: For any two vertices s, t the intersection of $L(s)$, $L(t)$, and the shortest s–t path P is nonempty. Then, the distance $\text{dist}(s, t)$ can be determined in linear (in the label size) time by evaluating $\text{dist}(s, t) = \min\{\text{dist}(s, u) + \text{dist}(u, t) \mid u \in L(s) \text{ and } u \in L(t)\}$. Also see Figure 2.11 for an illustration of this labeling method. Note that in [AFGW10] the bound $\mathcal{O}(\Delta h \log D)$ is achieved by a theoretical algorithm that computes labels according to small shortest path covers at different scales.

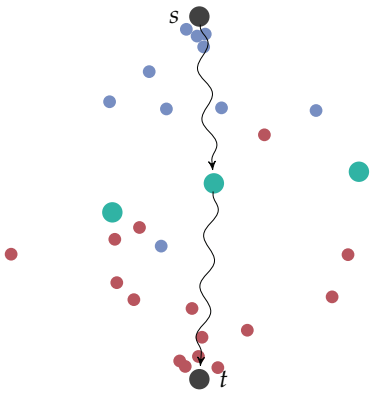


Figure 2.11. Illustrating hub labels of vertices s (blue) and t (red). The intersection is drawn green.

A practical (and highly engineered) implementation of the labeling scheme is

Hub Labels [ADGW11]. Essentially, the label of vertex u is defined by the (upward) search space of a Contraction Hierarchy query from u, but with suboptimal entries removed. At the time of writing, this algorithm is the fastest method for exact point-to-point queries in road networks: After roughly 2.5 h of preprocessing, in which it produces 21.3 GiB of data, queries can be answered in about 0.25 µs on average. Note that this is within only a factor of five of the memory access time on the considered machine. Beyond different implementations that trade query time for space [ADGW11], several optimizations that obtain smaller labels in less time [ADGW12] and that efficiently compress the labels (yielding the HLC algorithm) [DGW13], exist. Moreover, due to the very simple query algorithm, Hub Labels can even be implemented on top of relational databases in SQL [ADF+12].

Compressed Path Databases. The final method we review in this section is Compressed Path Databases (CPD) [Bot11]. Originally developed in the context of pathfinding in game maps, the technique has been recently adapted to road networks [BH13]. Its goal is to efficiently store all-pairs shortest path information, such that the shortest *path* can be retrieved quickly during the query. It therefore maintains, for each vertex $u \in G$, a label $L(u)$ that stores the *first move* (or first to u incident arc) of the shortest path toward *every* (other) vertex v of the graph. The query then starts at the source vertex s and scans $L(u)$ for t, which immediately yields the first arc (s, u) of the shortest path (to t). The algorithm then recurses on u until it reaches t.

Storing, for each vertex, the first arc of the shortest path to every other vertex explicitly, results in $\mathcal{O}(|V|^2)$ amount of data, which is prohibitive. Therefore, in [BH13] the data is compressed in a lossless fashion, based on the intuition that vertices of the same geographic region are likely to share the same first move from vertex u. The algorithm groups vertices that share the same first move into nonoverlapping geometric rectangles, and it only stores those with u. Further compression techniques include list trimming (implicitly storing the most frequent first move as a default), run length encoding, and sliding window compression on the rectangles.

Note that CPD can be seen as a hybrid goal-directed and table-based technique which shares some similarities with Geometric Containers [WWZ05]. However, an advantage of CPD over Geometric Containers (and in fact also other techniques) is that the first arc of the shortest path is returned immediately at the beginning of the query. In contrast, with Dijkstra-based methods the first arc of the shortest path is usually not known before the end of the algorithm's execution.

2.1.6. Combinations

Besides individual speedup techniques, systematic combinations of them have been studied as well [SWW00, HSWW06, BDS+10]. We briefly recap them in the following.

Basic Combinations. In [HSW04, HSWW06] the following combinations for some of the early speedup techniques have been considered. Goal-directed (A*) search and bidirectional search, A* search and multilevel overlay graphs, A* search and Geometric Containers (using bounding boxes), bidirectional search and multilevel overlay graphs, as well as, bidirectional search and Geometric Containers. The conducted experiments indicate that the combination of A* with multilevel overlay graphs has the best performance on road networks.

Note that most of the combinations work out of the box, though some, such as bidirectional geometric containers, may require additional auxiliary data. Moreover, the combination of A* with bidirectional search requires a careful (quite conservative) adaption of the stopping criterion, since otherwise queries may be incorrect [Poh69]. Unfortunately, this results in poor performance [KK97]. Therefore, in [HSWW06] the backward search uses the same potentials as the forward search. As a result, it is not goal-directed (toward s), but a stronger stopping criterion can be applied.

Moreover, in [SWW00] Geometric Containers, multilevel overlay graphs, and A* have been combined. However, they were only evaluated on railway networks, on which speedups in the order of 60 have been observed.

REAL. The *REAL* algorithm combines Reach, ALT and bidirectional search [GKW09]. Recall that ALT uses A* search with landmarks and the triangle inequality. To enable the stronger stopping criterion of bidirectional search, REAL combines the forward and backward potential functions π_f and π_r to obtain feasible potentials via $(\pi_f - \pi_r)/2$ (forward search) and $(\pi_r - \pi_f)/2$ (backward search). Moreover, a variant of the algorithm uses *reach-aware landmarks*: Landmarks and their distances are only precomputed for vertices with high reach values, which drastically reduces space consumption.

Core-ALT and HH*. Recall that the ALT algorithm [GH05] precomputes landmark distances for all vertices and landmarks in the graph, which results in a very high space consumption. This is remedied by Core-ALT [BDS+10, DN12]: It first computes an overlay graph for a (small) subset (e. g., 1 %) of important vertices, which is also called *core graph*. Core vertices are determined by, e. g., selecting the top most important vertices from a contraction hierarchy [GSSV12]. Landmark selection and their distance computation is then restricted to this core graph. The query works in two phases: The first runs a bidirectional search from s and t (which are possibly not in the core), until all branches of the shortest path trees are covered by core vertices. The second phase then runs ALT between these *entry* and *exit* vertices restricted to the core. Note that if t is not part of the core, the query must first determine the closest (to t) core vertex, which is then used as proxy in the triangle inequalities of ALT. Speedups of this method are less, if compared to other combinations. However, (Core-)ALT is very robust with respect to the input [BDW11] and can also be applied

in dynamic [DW07] and time-dependent [DW09b, DN12] scenarios

Using Highway Hierarchies [SS12a] (instead of Contraction Hierarchies) together with ALT results in the *HH** algorithm [DSSW09b]. Similarly to Core-ALT, landmarks and distances are only computed for important vertices and the query also works in two phases, where the first does not utilize goal-directed search.

ReachFlags. The *ReachFlags* method [BDS+10] combines Reach with Arc Flags. The preprocessing algorithm first computes (approximate) reach values for all vertices in G [GKW07]. In a second step, it extracts the subgraph H induced by all vertices whose reach value exceeds a certain (tuning) parameter. Arc-Flags are then only computed for the restricted subgraph H. The query runs, similarly to Core-ALT, two separate phases: The first utilizes a regular Reach query on G until all branches of the shortest path trees are covered by vertices from H. The second phase runs a combined Reach and Arc Flags query between these entry and exit vertices of H.

SHARC. The *SHARC* algorithm [BD09], which stands for *sh*ortcuts with *arc* flags, combines the computation of shortcuts with multilevel arc flags. The preprocessing algorithm first determines a partition of the graph and then computes shortcuts and arc flags in turn. Shortcuts are obtained by contracting unimportant vertices with the restriction that shortcuts never span different cells of the partition. The algorithm then computes arc flags such that, for each cell C, the query only uses a shortcut arc if and only if the target vertex is not contained in C. This results in an algorithm that is unidirectional *and* hierarchical: Arc flags not only guide the query toward the target, but also vertically across the hierarchy (of contracted vertices). This makes SHARC an excellent algorithm for scenarios where a backward search is prohibitive, such as in time-dependent route planning [Del11]. In addition, extensions of SHARC exist that reduce space consumption [BDGW10] and compute Pareto paths with respect to several optimization criteria [DW09a].

CHASE. Combining Contraction Hierarchies with Arc Flags results in the *CHASE* algorithm [BDS+10]: During preprocessing, a regular contraction hierarchy is computed and the search graph that includes all shortcuts is assembled. The algorithm then extracts a subgraph H from the search graph that is induced by the top k vertices of highest rank (with respect to the contraction order). Bidirectional arc flags (and the partition) are finally computed on the restricted subgraph H. The query runs, similarly to ReachFlags, two phases. The first performs a regular (bidirectional) Contraction Hierarchies search until the subgraph H covers all branches of the forward and backward shortest path trees. The second phase continues the Contraction Hierarchies query but also utilizes the arc flags. Arc Flags accelerate Contraction Hierarchies by about a factor of 10 with little additional overhead in space consumption [BDS+10].

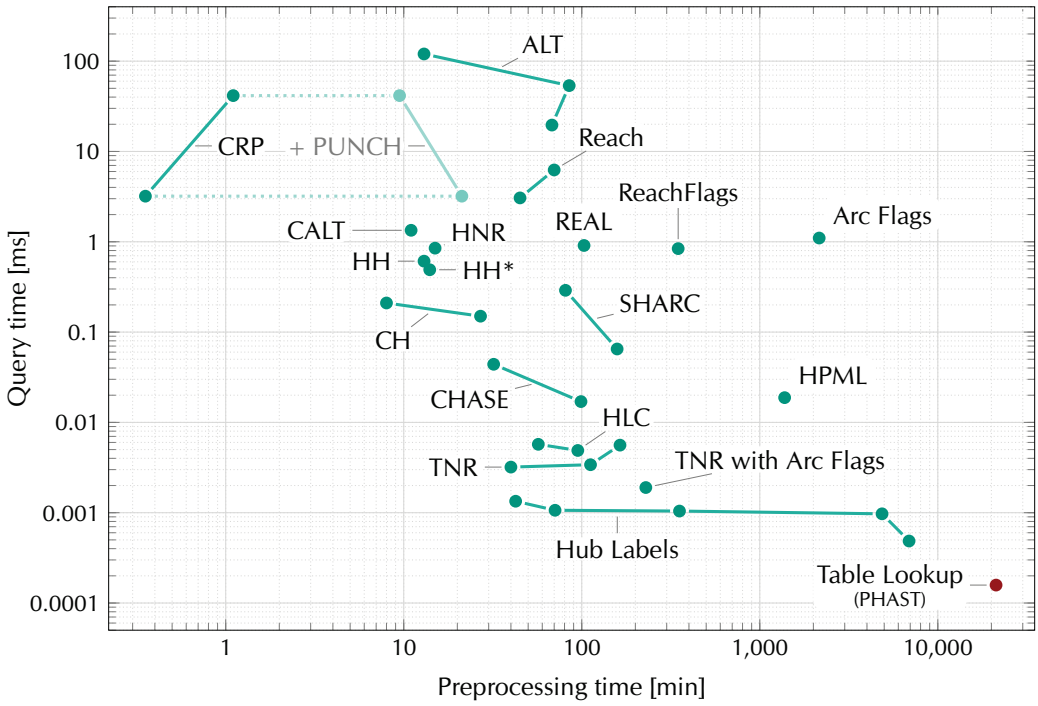


Figure 2.12. Preprocessing and average query time performance for algorithms with available experimental data on the European road network (and travel time metric). Connecting lines indicate different tradeoffs for the same algorithm. Figures for Reach, HH, HNR, ALT, Arc Flags, REAL, HH*, SHARC, CALT, CHASE, ReachFlags and TNR + AF are taken from [BDS+10]. Figures for TNR are taken from [BDS+10] and [DGW13]. Figures for Hub Labels are taken from [ADGW12] and [DGW13]. The figure for HPML stems from [ADGW11]. Contraction Hierarchies (CH) figures are taken from [GSSV12]. CRP (and the corresponding PUNCH) figures are taken from [DGPW14]. Note that CRP uses a more realistic graph model that also includes turn costs. Finally, the table lookup figure is based on the time of a single memory access and the precomputation time of $|V|$ shortest path trees using PHAST [DGNW13]. We remark that some preprocessing times, e. g., of Arc Flags and CHASE, could be accelerated by also using PHAST. Here, all figures correspond to a sequential execution of the algorithm and are (if necessary) scaled to a common machine (AMD Opteron 2218 with 2×1 MiB L2 cache). The figure is inspired by [Som12].

Transit Node Routing with Arc Flags. The final combination we survey is called *Transit Node Routing with Arc Flags* (TNR+AF) [BDS+10]. Recall that the bottleneck of the TNR query is performing the table lookups between pairs of access nodes from $A(s)$ and $A(t)$. To reduce the number of lookups, TNR+AF's preprocessing decomposes the set of transit nodes T into k regions. Then, for any vertex s and each of its access nodes $u \in A(s)$, it encodes in a k-bit vector to which regions the vertex u is important. More precisely, it sets the i-th bit to true, if there is a shortest path to a transit node v of region i that uses u as access node for a shortest path from s. The

query then only considers those access nodes from s that have their bits set with respect to the regions of $A(t)$ (and vice versa). TNR+AF requires quite some space to store the flags (a factor of two more compared to plain TNR), however, query times of only 1.9 µs on the road network of Europe make TNR+AF the fastest available combination [BDS+10]. Note that only Hub Labels has lower query times.

2.1.7. Theoretical Results

Besides the great amount of experimental work, also some theoretical results on route planning in road networks exist. We briefly recap the most important and group them by results on preprocessing complexity and query performance bounds.

Preprocessing Complexity. Query algorithms of almost all (previously in this section) surveyed methods compute provably optimal paths. On the other hand, the preprocessing phase often leaves some degree of freedom, which is usually filled in a heuristic way. For example consider Contraction Hierarchies [GSSV12]. Here, the vertex order determines the number of added shortcuts and, as such, the performance of the query algorithm. More generally, one may ask how to perform exact preprocessing such that the (average or worst case) query time is minimized. Note that, since query time is hard to analyze, one often uses *search space* as a proxy.

Shortcuts are an ingredient to many hierarchical speedup techniques, such as SHARC [BD09]. Deciding whether a fixed number k of shortcuts can be added to a graph, such that the search space size decreases by at least a constant c on average, is an NP-hard problem. However, a greedy factor-k approximation algorithm exists [BDD+12]. Unfortunately, it turns out that optimal preprocessing is also NP-hard for the following methods [BCK+10]: ALT (with respect to landmark selection), Arc Flags (with respect to the partition), SHARC (with respect to the shortcuts), Multilevel Overlay Graphs (with respect to the separator), and Contraction Hierarchies (with respect to the vertex order). Finally, in [BBRW13] preprocessing of Arc Flags is analyzed in more detail and on restricted graph classes, such as paths, trees, and cycles. It turns out that determining optimal partitions (which minimize the query's search space) is already NP-hard for binary trees.

Performance Bounds. Besides complexity, theoretical performance bounds for query algorithms, which aim to explain their excellent practical performance, have also been considered. However, proving better running time bounds than that of Dijkstra's algorithm seemed long challenging. In fact, it is not hard to construct inputs for which most algorithms admit no speedup [AFGW10]. Therefore, a (theoretical) explanation for the great practical performance of these speedup techniques can only be achieved in conjunction with a formalization of a suitable property that defines some key features of real world road networks.

In a seminal paper by Abraham et al. [AFGW10], such a graph property, called *Highway Dimension*, is proposed. Roughly speaking, a graph has highway dimension h, if at any scale r and any vertex u, in the ball $B_r(u)$ of radius r around u, all shortest paths of length $r/2$ can be covered with at most h vertices. Depending on h, bounds for Reach, Contraction Hierarchies, and the labeling method exist that only depend on h, the graph's diameter D, and its maximum degree Δ [ADF+11]. More precisely, after running a polynomial time preprocessing routine, which adds $\mathcal{O}(h \log h \log D)$ shortcuts to G, Reach and Contraction Hierarchies run in time $\mathcal{O}((\Delta + h \log h \log D)(h \log h \log D))$, and the labeling algorithm runs in time $\mathcal{O}(\Delta + h \log h \log D)$. Note that in [AFGW10], it is conjectured that road networks have small highway dimension. These bounds, in particular the polynomial time preprocessing algorithm, are achieved in [ADF+11] by connecting the notions of highway dimension and VC-dimension [VC71].

Besides these results, Rice and Tsotras [RT12] analyze the (heuristic variant of the) A* algorithm and obtain bounds on the search space size that depend on the underestimation error of the potential function. Also, maintaining and updating multilevel overlay graphs have been theoretically analyzed in [BCD+08]. For Transit Node Routing, instance-based lower bounds on the size of a transit node set that must cover shortest paths at a certain scale, are given in [EF12]. Regarding the labeling method, bounds on the label size for different graph classes are given in [GPPR04], and approximation algorithms that compute small labels have also been studied [CHKZ03, BGGN13].

Finally, Contraction Hierarchies have been analyzed in [BCRW13] by connecting them to the notions of filled graphs [Par61] and elimination trees [Sch82]. Nested dissections of G imply vertex orders for CH, such that for graphs of treewidth k the search space of CH is bounded by $\mathcal{O}(k \log |V|)$. Similarly, for minor-closed graph classes with balanced $\mathcal{O}(\sqrt{|V|})$-separators, the search space is bounded by $\mathcal{O}(\sqrt{|V|})$.

2.2. Journey Planning in Public Transit Networks

This section surveys related literature on journey planning in (schedule-based) public transit networks. In this scenario, the input is given by a timetable. Roughly speaking, a timetable consists of a set of stops (or stations, platforms, etc), a set of routes (such as bus lines), and a set of trips. Trips are individual vehicles that visit the stops along a certain route at a specific time of the day. (See Section 4.1 for a precise definition.)

A key difference to road networks is that public transit networks are inherently *time-dependent*: Certain segments of the network can be traversed at specific, discrete points in time, only. As such, the first challenge concerns modeling the timetable appropriately in order to enable the computation of journeys. While in road networks the objective to compute a single shortest path (i. e., quickest journey) is often sufficient, in public transit networks more involved problems (e. g., taking several

optimization criteria into account simultaneously) are important. We address them in a separate modeling Section 2.2.1.

Work on accelerating queries for efficient journey planning started by Schulz et al. in [SWW99]. Since then, a great amount of algorithms were developed that concern—besides accelerating the query—extended scenarios that incorporate delays, compute robust journeys, or optimize additional criteria, such as monetary cost.

Real world journey planning systems include ARIADNE [BS88], which was in use by the German railways and later superseded by HAFAS (HaCon Fahrplan-Auskunfts-System) [HaC] from HaCon [HaC84]. Another commercial system, especially used by many local transit agencies, is EFA (Elektronische Fahrplanauskunft) from Mentz Datenverarbeitung [Men]. The system TRAINS [TS88, TS91] has been used by the Dutch railways as a prototype. Finally, the Transfer Patterns algorithm [BCE+10] is currently in use by Google Transit [Goo10] and RAPTOR (which was developed in this thesis; see Section 4.6) is currently in use by OpenTripPlanner [Ope12].

2.2.1. Modeling

The first challenge is to model the timetable in order to enable algorithms that compute optimal journeys. Since the shortest path problem is well understood in the literature, it seems natural to build a graph $G = (V, A)$ from the timetable such that shortest paths in G correspond to optimal journeys. Two main approaches exist: The *time-expanded approach* and the *time-dependent approach*. We review them in the following and also look at the type of problems one is often interested to solve. Besides individual publications, there is an excellent overview article by Müller-Hannemann et al. [MSWZ07]. Also, see Sections 4.2 and 4.3 for more details.

Time-Expanded Model. Recall that the input, i. e., the timetable, is time-dependent by definition (cf. Section 4.1). Based on the fact that these time-dependent *events* (e. g., a vehicle departing at a stop) happen at *discrete* points in time, the idea of the *time-expanded* model is to build a space-time graph (often also called an event graph) [PS98] that "unrolls" time. Roughly speaking, the model creates a vertex for every event of the timetable, connects subsequent events in direction of time flow by arcs. In [Möh99, SWW00] a basic version of the model is used to compute shortest paths: For every departure and arrival event, it contains a vertex, and each subsequent departure and arrival event is connected by a *connection arc*. To enable transfers between vehicles, all vertices at the same stop are (linearly in order of time) interlinked by *transfer arcs*. Müller-Hannemann and Schnee [MW01] extend the model to distinguish trains (to optimize the number of taken transfers during query) by subdividing each connection arc by a new vertex, and then interlinking the vertices of each trip (in order of travel). In [PSWZ08] the time-expanded graph is extended to incorporate *minimum change times* (given by the input) that are required as buffer when changing trips at a station: In their *realistic* model they introduce an additional *transfer vertex* per departure

event, and connect each arrival vertex to the first transfer vertex that obeys the minimum change time constraints. Finally, the model has been further engineered in [DPW09b] to reduce the number of "redundantly" explored arcs during query. Also see Section 4.3.2 for details.

Time-Dependent Model. The main disadvantage of the time-expanded model is that the resulting graphs are fairly huge [PSWZ04a]. The *time-dependent approach*, in contrast, produces significantly smaller (in terms of number of vertices and arcs) graphs by not unrolling the timetable. Instead, time-dependencies are encoded by *travel time functions* on the arcs that map departure times to travel times. Evaluating the cost of an arc then depends on the time, at which it is traversed. A general analysis of time-dependent shortest paths under various waiting constraints is conducted by Orda and Rom [OR90, OR91]. It turns out that the shortest path problem can be efficiently solved if travel time functions are nonnegative and FIFO, i. e., waiting never pays off.

In the context of computing optimal journeys in public transit networks, the time-dependent approach has been proposed in [BJ04]. Here, vertices correspond to stops, and an arc is added from u to v, if there is at least one trip serving the corresponding stops in this order. Precise departure and arrival times are encoded by the associated travel time function of the arc (u, v). In [PSWZ08] this basic model has been further extended to enable minimum change times. Roughly speaking, it creates, for each stop p and each route that serves p, a dedicated *route vertex*. Route vertices at p are connected to a common *stop vertex* by arcs with constant cost depicting the minimum change time of p. Trips are distributed among *route arcs* that connect the subsequent route vertices of a route. In addition, a model that handles variable change times that allow arbitrary minimum change times between pairs of routes is also presented in [PSWZ08]. See Section 4.3.3 for details.

Problem Variants. In road networks an obvious problem is to compute the quickest route (that is, the shortest path). Hence, much research focused on this task. For public transit networks, however, several problems arise that are equally important. We briefly review them in the following.

The simplest is the *earliest arrival problem*, which has been first considered by Schulz et al. [SWW00]. Given source and target stops p_s, p_t and departure time τ, it asks for a journey that departs p_s no earlier than τ and arrives at p_t as early as possible.

The *range problem* (also called *profile problem*) was first considered in the context of public transit networks by Nachtigall [Nac95]. It drops the departure time from the input. Instead, it asks for a set of journeys of minimum travel time that all depart within a given time range (possibly the full day).

Both the earliest arrival and the range problems only consider (arrival or travel) time as criterion. However, in public transit networks other criteria, such as the

number of transfers, are just as important. Therefore, Müller-Hannemann and Weihe [MW01] consider the *multicriteria problem*. Given source and target stops p_s, p_t and departure time τ as input, it asks for a (maximal) Pareto set $\mathcal{J}$ of nondominating journeys with respect to the considered optimization criteria. Thereby, a journey J_1 dominates journey J_2, if J_1 is better or equal in all criteria than J_2. Further variants of the problem relax or strengthen these domination rules [MW01, MS07].

2.2.2. Search Algorithms without Preprocessing

This section discusses algorithms that solve one of the aforementioned problems, without yet employing a preprocessing phase. We group them by the respective problem they solve. Note that these algorithms can instantly be used in dynamic scenarios that include delays, route changes, or trip cancellations.

Earliest Arrival Problem. Computing earliest arrival queries on the time-expanded model can be achieved by Dijkstra's algorithm [SWW00]. The algorithm is initialized with the vertex that corresponds to the smallest (in time) event of the source stop p_s that occurs after τ (in the realistic model, a transfer vertex must be selected). The first scanned vertex associated with the target stop p_t then represents the earliest arrival s–t journey. On time-dependent graphs Dijkstra's algorithm can be augmented to compute shortest paths [CH66, Dre69], if the cost functions are nonnegative and FIFO [OR90, OR91]. The only modification is the following: Whenever the algorithm scans an arc (u, v), its cost is evaluated at time $\tau + \text{dist}(s, u)$. Note that the algorithm retains the label-setting (cf. Section 2.1.1) property, i. e., vertices are scanned at most once. In the time-dependent public transit model, the query is run from the stop vertex corresponding to p_s and the algorithm may stop as soon as it extracts p_t from the priority queue.

Recently, a new approach to compute earliest arrival queries, called *Connection Scan Algorithm* (CSA) [DPSW13], has been developed. It is not graph-based and uses no priority queue. Instead, it organizes the connections of the timetable in a single array, sorted by departure time. The query then only scans over this array once, which turns out to be very efficient in practice.

Range Problem. The range problem can be solved on the time-dependent model by variants of Dijkstra's algorithm. The first variant, which has been studied in [Nac95, Dea99], maintains at each vertex u of the graph a travel time function (instead of a scalar label) depicting the optimal travel times from s to u for the considered time range. Whenever the algorithm relaxes an arc (u, v), it *links* the travel time function associated with u to the (time-dependent) cost function of the arc (u, v). The resulting function is then *merged* into the (tentative) travel time function associated with the vertex v. The algorithm loses the label-setting property, since travel time functions cannot be totally ordered. As a result the algorithm may reinsert vertices into the

priority queue whenever it finds a journey that improves the travel time function of an already scanned vertex. Another algorithm, considered in [Bau12], exploits the fact that trips depart at discrete points in time. It, therefore, does not propagate the full function when it relaxes an arc, but considers each *connection point* that represents a discrete departure event. By these means, the number of redundant vertex scans can be significantly reduced.

Finally, the Connection Scan Algorithm has also been extended to the range problem in [DPSW13]. It uses the same array of connections, ordered by departure time, as for earliest arrival queries. It still suffices to scan this array once, even to obtain optimal journeys to all stops of the network.

Multicriteria Problem. The multicriteria problem received quite some attention in the literature. Computing Pareto sets of shortest paths in (general) graphs can be done by extensions of Dijkstra's algorithm (see [EG02] for a survey on multicriteria combinatorial optimization). More specifically, the *Multi-Label-Correcting* (MLC) algorithm [Han79, Mar84, The95, Möh99] extends Dijkstra's algorithm by keeping, for each vertex, a *bag* of nondominated labels. Each label is represented as a tuple, with one entry per optimization criterion. The priority queue maintains labels instead of vertices and orders them lexicographically. In each iteration, it extracts the minimum label L and scans the incident arcs $a = (u, v)$ of the vertex u associated with L. It does so by adding the cost of a to L and then merging L into the bag of v, eliminating possibly dominated labels on the fly.

On the time-expanded model this algorithm has been considered in a framework called PARETO by Müller-Hannemann and Schnee [MS07]. They optimize arrival time, ticket cost, and number of transfers. On the time-dependent model, computing Pareto sets of journeys for arrival time and number of transfers has been considered in [PSWZ08]. Disser et al. [DMS08] propose three optimizations to MLC that reduce the number of queue operations: Hopping reduction, label forwarding, and dominance by early results (also called target pruning in this thesis).

In [Han79] it is observed that Pareto sets may contain exponentially many solutions, even for the restricted case of two optimization criteria. To accelerate the query, one can compute approximate solutions, for example, by relaxing domination. In particular, $(1 + \epsilon)$-Pareto sets have provable polynomial size [PY00] and can be computed efficiently [Lor84, Whi86, TZ06]. This approach has been applied to public transit journey planning in [MS07]. For the case of optimizing earliest arrival time and number of transfers, the *Layered Dijkstra* (LD) algorithm [BJ04, PSWZ08] is also more efficient: Given an upper bound K on the number of transfers, it copies the graph into K layers, rewiring transfer arcs to point to the next higher level. It then suffices to run a time-dependent (single criterion) Dijkstra query from the lowest level to obtain Pareto sets.

2.2.3. Speedup Techniques

This section gives an overview on preprocessing-based speedup techniques for journey planning in public transit networks. Most research focused on *adapting* existing methods from road networks. This seemed quite natural because of their exceptional performance on those networks (see Figure 2.12). Unfortunately, the speedups observed in public transit networks are several orders of magnitude lower. This is to some extent explained by the quite different structural properties of transit and road networks [Bas09]. Thus, developing efficient preprocessing-based methods for public transit remains a challenging goal.

Some road network methods were tested on public transit graphs without performing realistic queries (i. e., according to one of the problems from Section 2.2.1). In [HSWW06] basic combinations of bidirectional search, goal directed search, and Geometric Containers have been evaluated on a simple stop graph (with average travel times). In [BDW11] bidirectional search, ALT, Arc Flags, Reach, REAL, Highway Hierarchies, and SHARC were evaluated on time-expanded graphs. Moreover, Core-ALT, CHASE, and Contraction Hierarchies were evaluated in [BDS$^+$10] (also on time-expanded graphs). Note that both [BDW11] and [BDS$^+$10] run point-to-point queries between arbitrary vertices (events) of the graph.

A* Search. Basic goal-directed A* search [HNR68] has been considered on time-dependent graphs in the context of road networks in [Fli04]. On public transit networks, it has been applied to the time-dependent model in [DMS08] (in the context of multicriteria optimization). Here, lower bounds for each vertex u to the target stop p_t are determined (before the query) by running a backward search (from p_t) using the (constant) lower bounds of the travel time functions as arc cost.

ALT. The (unidirectional) ALT [GH05] algorithm has been adapted to both the time-expanded [DPW09b] and the time-dependent model [Del11] for computing earliest arrival queries. In both cases, landmark selection and distance precomputation is performed on an auxiliary stop graph: Vertices correspond to the stops of the timetable, and an arc is added between two stops p_i, p_j, if there is a trip that serves p_i and p_j (in this order) without intermediate stop. Arc costs depict lower bounds on the travel time between their incident stops.

Geometric Containers. Geometric containers [SWW00, WWZ05] have been extensively tested on the time-expanded model for computing earliest arrival queries. (In fact, they were developed in the context of this model.) In [SWW00] the algorithm has been evaluated using angular sectors as container, while more sophisticated containers have been tested in [WWZ05]. The latter work concludes that bounding box containers perform best.

Arc Flags and SHARC. Arc Flags [Lau04] have been adapted to the time-expanded model as follows [DPW09b]: First, the partition is computed on the stop graph, which is defined equally to ALT. Then, for each boundary stop p of cell C, and each of its arrival vertices, a backward search in the time-expanded graph is performed. It is observed in [DPW09b] that in public transit networks many paths of equal length exist between the same pair of vertices. This makes the consideration of appropriate tie breaking rules important. Furthermore, [DPW09b] combines Arc Flags with ALT and *Node Blocking*—a technique that avoids exploring redundant parts of the graph.

SHARC, which combines Arc Flags with shortcuts [BD09], has been tested on the time-dependent model with earliest arrival queries in [Del11]. Moreover, Arc Flags for the Multi-Label-Correcting algorithm (MLC) have been considered for computing full (i. e., using strict domination) Pareto sets regarding the criteria arrival time and number of transfers on a realistic time-dependent model that handles traffic days, train attributes, and minimum change times [BDGM09]. In time-dependent graphs, a flag must be set, if its arc is at least once during the day on a shortest path toward the respective cell [Del11]. In order to improve performance, one can use different sets of flags for different times of the day (e. g., every two hours). Beyond that, [BDGM09] combines Arc Flags with shortcuts (similarly to SHARC) to gain additional speedups and [BGM10] further exploits a property called event-dependent c-optimality. Combining all these optimizations, the total speedup of the algorithm is still below 15, from which it is concluded that "accelerating time-dependent multicriteria timetable information is harder than expected" [BDGM09].

Overlay Graphs. Using overlay graphs [SWW00, SWZ02] to accelerate queries has been—similarly to Geometric Containers— introduced in the context of public transit journey planning. In [SWW00] single level overlays are computed between "important" hub stations in the time-expanded model. Thereby, importance values are determined by the input. Multilevel overlay graphs based on vertex separators were developed in the context of time-expanded graphs in [SWZ02]. A systematic experimental study, which also includes time-expanded transit networks, is conducted in [HSW08].

Contraction Hierarchies. The Contraction Hierarchies algorithm [GSSV12] has been adapted to the realistic time-dependent model with minimum change times for computing earliest arrival and range queries [Gei10]. It turns out that simply applying the algorithm to the route model graph results in too many shortcuts to be practical. Therefore, contraction is performed on a condensed graph that contains only a single vertex per stop. Minimum change times are then ensured by the query algorithm, which must maintain multiple labels per vertex.

Transfer Patterns. A speedup technique specifically developed for public transit networks is called *Transfer Patterns* [BCE+10]. It is based on the observation that many optimal journeys share the same transfer pattern, i. e., the sequence of stops where a transfer occurs. These transfer patterns are precomputed for all pairs of stops during preprocessing. Then, given source and target stops p_s, p_t, the query quickly builds a search graph of (at least) the relevant transfer patterns to get from p_s to p_t. Note that arcs in this graph represent direct travel between transfers. Dijkstra's algorithm (or MLC) can then be applied to this significantly smaller search graph.

Precomputing transfer patterns between *all* pairs of stops turns out to be too expensive in practice. Therefore, a two levels approach (similarly to Transit Node Routing), first selects a subset of important hub stops (cf. transit nodes). Global transfer patterns are precomputed between these hub stops. Additionally, for each regular stop, local transfer patterns are computed toward (and from) its relevant hub stops (cf. access nodes). Unfortunately, preprocessing times are still impractical on continental networks. Therefore, one may trade optimality for a more practical preprocessing, which restricts the computation of local transfer patterns to at most three legs (two transfers). By these means, preprocessing times drop to slightly over 3 000 hours (on the large-scale transit network of North America), which then enables queries in the order of 10 ms (earliest arrival and multicriteria). The Transfer Patterns algorithm is currently in use by Google Transit [Goo10, BCE+10].

TRANSIT. Finally, Transit Node Routing [BFM+07, BFSS07, BFM09, SS09] has been adapted to public transit journey planning in [AW12]. Preprocessing of the resulting *TRANSIT* algorithm uses the (small) stop graph to determine a set of transit nodes (with a similar method as in [BFM09]), between which it maintains a distance table that contains sets of journeys with minimal travel time (over the day). Each stop p maintains, in addition, a set of access nodes $A(p)$, which is computed on the time-expanded graph by running local searches from each departure event of p toward the transit stops. The query then uses the access nodes of p_s and p_t and the distance table to resolve global requests. For local requests, it runs goal-directed A* search.

2.2.4. Extended Scenarios

Besides computing journeys according to one of the problems from Section 2.2.1, extended scenarios, e. g., incorporating delays, have been studied as well.

Uncertainty and Delays. Trains (and other means of transport) are often prone to delays in the real world. Thus, handling delays (and other uncertainty) is an important aspect of a practical journey planning system. Müller-Hannemann and Schnee [MS09] consider the online problem where delays, train cancellations, and extra trains arrive as a continuous stream of information. They present an approach

which quickly updates the time-expanded model to enable queries according to the new traffic situation. A realistic stochastic model that predicts how such delays propagate through the network is proposed in [BGMO11]. In particular, this model is evaluated using real (delay) data from Deutsche Bahn.

In [DMS08] the computation of *reliable* journeys is studied via multicriteria optimization. Thereby, the reliability of a transfer is defined by the probability that the particular transfer can be made successfully. Note that by this notion, transfers (with high chance of success) are also considered reliable, if no backup alternative for the (unlikely) case that the transfer fails exists.

Therefore, Dibbelt et al. [DPSW13] minimize the *expected arrival time*. Instead of journeys, their method outputs a *decision graph* depicting optimal instructions to the user at each point of their journey. Note that these instructions include the case that a connecting trip is missed. Interestingly, minimizing the expected arrival time implicitly also minimizes the number of transfers: Each "unnecessary" transfer introduces additional uncertainty which hurts the expected arrival time.

Finally, in [GKM+11, GKM+13] the computation of *robust* journeys is studied, considering both strict robustness (i. e., computing journeys that are always feasible for a given set of delay scenarios) and light robustness (i. e., computing journeys that are most reliable when given some extra slack time). Strict robustness turns out to be too conservative in practice, while the notion of light robustness seems more promising.

Night Trains and Fares. Explicitly computing overnight train journeys has been considered by Gunkel et al. [GMS07]. Interestingly, the optimization goals for such journeys are quite different from regular "daytime" journeys: From a customer's point of view, the primary objective is usually to have a reasonably long sleeping period. Moreover, arriving too early in the morning at the destination is often not desired as well. In [GMS07] two approaches to compute overnight journeys are presented. The first explicitly enumerates all overnight trains (which are given by the input) and computes, for each, the optimal feeding connections. The second runs multicriteria search with sleeping time as a maximization criterion.

Finally, several tariff schemes have been analyzed in [MS06]. Some of them were also integrated as an optimization criterion (cost) into a multicriteria search algorithm (called MOTIS), which works on the time-expanded model. However, generally optimizing exact monetary cost is a challenging problem, since real world pricing schemes are hard to capture by a mathematical model [MS06]. (See also Section 4.6.6 where we optimize fare zones with our new McRAPTOR algorithm.)

2.3. Journey Planning in Multimodal Networks

This section surveys literature on journey planning in multimodal networks. Here, the general problem is to compute journeys that *reasonably* combine different modes of transport by a *holistic* approach. Transportation modes usually considered include (unrestricted) walking, (unrestricted) car travel, (local and long-distance) public transit, flight networks, and rental bicycle schemes. We emphasize that our definition of "multimodal" requires at least some diversity from the aforementioned transportation modes. Moreover, optimizing the choice (and sequence) of transportation modes should be an explicit ingredient of the algorithm. That is, computing, e. g., earliest arrival journeys that arbitrarily select the transportation modes bus, tram, and railway does not yet classify as multimodal journey planning (according to our definition). Also, these networks could essentially be represented by a single public transit timetable (cf. Section 2.2).

In fact, considering modal transfers explicitly in the algorithm is crucial in practice: The computed solutions should be *feasible*, i. e., not contain a sequence of transport modes which is impossible for the user to take (such as a private car between train rides). Ideally, even *preferences* of the user should be respected (e. g., some users may prefer a taxi over public transit at certain parts of the journey, others may not). See Section 5 for more details on these issues.

We organize this section in two parts: Modeling issues and search algorithms.

2.3.1. Modeling

This section presents important modeling issues arising in the context of multimodal journey planning.

Multimodal Networks. A general approach to obtain a multimodal network first builds an individual graph model of each considered transportation mode and then merges them to a multimodal graph, adding *link arcs* (or vertices) to enable modal transfers [Paj09, DPW09a, YL12]. In [Paj09, DPW09a] multimodal networks consisting of the following graph models are studied. Car travel and walking are both modeled as time-independent graphs, public transit networks are based on the realistic time-dependent model [PSWZ08], and for the flight network a dedicated flight model—which has been introduced in [DPWZ09]—is used. Beyond that, Kirchler et al. [KLPC11, KLC12] compute multimodal journeys where car travel is modeled as a time-dependent network in order to incorporate historic data on rush hours and traffic congestions. (See [DW09b] for an overview on time-dependent route planning in road networks.)

Combined Cost Functions. To avoid unreasonable combinations of transport modes, one may utilize penalties in the objective function of the algorithm. Often such

penalties are integrated into the objective function by linearly combining them with the primary optimization goal (usually travel time). In [AZC07] a linear programming approach with a linear objective function is presented that computes multimodal journeys. A multimodal journey planning algorithm, called TRANSIT [AW12], uses a linear utility function to incorporate travel time, ticket cost, and inconvenience of transfers. In [MS98] a combined network of unrestricted walking, unrestricted car travel, and public transit is considered. Journeys are optimized according to a linear combination of several criteria, which also handles user preferences.

Label-Constrained Shortest Paths. A quite elegant approach that guarantees computing journeys that obey certain transport mode constraints, is called *label-constrained shortest paths approach* [BJM00]. It defines an alphabet Σ of transport modes and each arc of the graph is labeled by the symbol from Σ that represents its respective transport mode. Then, given a language L over Σ as additional input to the query, any journey (path) must obey the constraints imposed by the language L. More precisely, the concatenation of the labels along the path must satisfy L. The problem of computing *shortest* label-constrained paths becomes tractable for *regular* languages [BJM00]. Fortunately, regular languages suffice to model reasonable transport mode constraints in multimodal journey planning [BBJ+02, BBH+08]. Often, even restricted classes of regular languages are considered as constraints, for example, languages that impose a hierarchy of transport modes [BBM06, Paj09, DPW09a, KLPC11, KLC12, YL12], or Kleene languages that can only globally exclude (and include) certain transport modes [RT10].

Note that label-constrained shortest paths are also useful in other scenarios, such as in database query optimization [MW95].

Multicriteria Optimization. While label-constraints are useful to define feasible journeys, computing the (single) shortest label-constrained path may be disadvantageous for two reasons. First, the user has to define the constraints, for which he has to know the characteristics of the particular transportation network, and, second, no alternative journeys that differently combine the available transportation modes are computed. To obtain a set of diverse alternatives, multicriteria optimization has been considered: In [YL12] sets of journeys are obtained which are prioritized according to the preferred transport modes (given as user input). In [EL11] Pareto sets that optimize several criteria are computed. Unfortunately, these sets can get fairly large, containing many solutions with insignificant tradeoffs in the considered criteria [BBS13]. This makes it necessary to identify the most significant solutions of the Pareto set in a postprocessing step.

2.3.2. Search Algorithms

This section discusses multimodal journey planning algorithms. Thereby, most work focused on the label-constrained shortest path problem, for which also some speedup techniques, which employ a preprocessing phase, exist. Note that the multicriteria problem, can be solved—equivalently to journey planning in public transit networks—by the MLC algorithm by applying it to the (integrated) multimodal graph.

Label-Constrained Shortest Paths. In [BJM00] it is proven by construction that the label-constrained shortest path problem is solvable in deterministic polynomial time. The algorithm, called *label-constrained shortest path problem Dijkstra* (LCSPP-D), first builds a product network of the input (i. e., the multimodal graph) and the (possibly nondeterministic) finite automaton that accepts the regular language L. Then, for source and target vertices s, t (referring to the original input), Dijkstra's algorithm is run on the just-constructed product network between all vertices that correspond to s and the initial states of the automaton and those which correspond to t and the final states of the automaton. A followup experimental study that evaluates this algorithm using linear regular languages (a special case) has been conducted in [BBJ+02].

In [Paj09] the LCSPP-D algorithm has been combined with time-dependent Dijkstra [CH66] to compute journeys in multimodal networks that contain a time-dependent subnetwork. The adaption of basic ingredients (to speedup techniques in road networks; cf. Section 2.1), such as bidirectional search [Dan62], ALT [GH05], Arc Flags [Lau09, HKMS09], and shortcuts [SWW00, SS05, GSSV12], has been analyzed in [Paj09] as well. Also, some basic speedup techniques, such as bidirectional search [Dan62], A* [HNR68], and the Sedgewick-Vitter Heuristic [SV86] have been evaluated in the context of multimodal journey planning in [Hol08, BBH+09].

Access-Node Routing. A speedup technique developed for the label-constrained shortest path problem (LCSPP) is called *Access-Node Routing* (ANR) [DPW09a]. It handles *hierarchical languages* where walking and car travel is restricted to the beginning and end of the journey. It works similarly to Transit Node Routing [BFM+07, BFSS07, BFM09, SS09] and precomputes for each vertex u of the road (walking and car) network its relevant set of entry (and exit) points (*access nodes*) to the public transit and flight networks. More precisely, for any shortest path P originating from vertex u (of the road network) that also uses the public transit network, the first vertex v of the public transit network on P must be an access node of u. Having computed these access nodes (with their corresponding distances), the query may skip over the road network by running a multi-source multi-target algorithm on the (much smaller) transit network between the access nodes of s and t, returning the journey with earliest combined arrival time.

To further reduce preprocessing space and time, Access-Node Routing has been combined with contraction, resulting in a method called *Core-Based ANR* [DPW09a].

Similarly to Core-ALT [BDS+10, DN12], it precomputes access nodes only for road vertices in a much smaller core (overlay) graph. The query algorithm must, thus, first (quickly) determine the relevant core vertices of s and t (i. e., core vertices covering the branches of the shortest path trees rooted at s and t), before it commences with a multi-source multi-target ANR query between these core vertices.

Access-Node Routing has been evaluated on multimodal networks of intercontinental size that include—besides walking and car travel—public transit and flights. It achieves query times in the order of milliseconds, however, preprocessing performance strongly depends on the density of the public transit and flight networks [DPW09a] (see also Section 5.3.5). Moreover, the regular language is determined during preprocessing and can, thus, no longer be specified as an input to the query without losing optimality.

State-Dependent ALT. Another multimodal speedup technique for LCSPP is *State-Dependent ALT* (SDALT) [KLPC11]. It augments the ALT algorithm [GH05] based on the idea that lower bounds from a vertex u may vary significantly depending on the current state q of the automaton (corresponding to the considered regular language) with which u is scanned. Thus, just precomputing a single landmark distance value per vertex (like ALT) may result in poor bounds. In contrast, SDALT uses the automaton to precompute state-dependent distances, providing lower bound values per vertex *and* state. To further improve query performance, SDALT has also been extended to handle incorrect lower bounds, which guides the search stronger toward the target. To still maintain correctness, the query uses a label-correcting algorithm (instead of Dijkstra's algorithm), which may scan vertices multiple times [KLC12].

SDALT has been evaluated on a highly realistic multimodal network covering the Île-de-France area (containing Paris) [KLPC11, KLC12], resulting in speedup factors of up to 30. The considered transport modes include rental and private bicycles, public transit, walking, and a time-dependent road network for car travel. Note that SDALT, like ANR, also predetermines the regular language constraints during preprocessing.

Contraction Hierarchies. Contraction Hierarchies [GSSV12] have been adapted to a restricted version of LCSPP that considers *Kleene Languages* [RT10, GRST12]. Note that Kleene languages are a relatively strong restriction of regular languages: They can specify *which* transport modes a journey may contain, but not the *sequence* in which they are allowed to appear. However, the algorithm presented in [RT10] allows arbitrary Kleene languages specified as a *query* input.

Therefore, the Contraction Hierarchies preprocessing is adapted as follows: Each arc $a \in A$ maintains a set $L(a)$ of labels from L, initially only containing the symbol (from Σ) depicting the transport mode represented by a. Whenever the algorithm contracts a vertex v, it must determine, for each pair $(u, v), (v, w)$ of arcs, whether a shortcut from u to w is necessary to preserve distances. It does so by running

a (modified) local search from u which excludes arcs whose labels contain symbols from the set $L \setminus (L(u,v) \cup L(v,w))$. If the local search failed to find a shorter path than the combined length of $\ell(u,v) + \ell(v,w)$, the shortcut (u,w) must be added, and its associated label $L(u,w)$ is set to $L(u,v) \cup L(v,w)$. Note that by these means parallel edges that contain different subsets of Σ in their labels may exist [RT10]. The algorithm has been further extended in [GRST12] to handle even more flexible edge restrictions (such as vehicle height) as a query input.

Chapter 3

Fundamentals

Within this chapter we introduce fundamental notion that is used throughout this work. In particular, we address graph theory, partitions, regular languages, and finite automata. While we introduce most concepts that are relevant to this work, we assume familiarity with basic set theory, predicate logic, and basic tools for algorithm analysis, such as Landau notation. Also see [CLRS01].

3.1. Graph Theory

Transportation networks can be modeled as graphs. Therefore, graphs are at the very heart of this thesis. We introduce basic notions of graph theory in the following. Moreover, in time-*dependent* networks (such as those arising in public transit) costs can be reflected by special functions, which are also introduced here.

Graphs. A *graph* $G = (V, A)$ consists of a set V of *vertices* and a set A of *arcs*. Usually, the graphs we work with in this thesis are *directed*, that is, $A \subseteq V \times V$ consists of *ordered pairs* of vertices. For two vertices $u, v \in V$, we say there is an arc from u to v in G, if and only if $(u, v) \in A$ holds. For short, we sometimes also write uv to refer to the arc (u, v). Given an arc $a = (u, v)$, we call u the arc's *tail* and v the arc's *head*. Note that (u, v) and (v, u) are different arcs. For the case this distinction is not required, we consider *undirected* graphs $G = (V, E)$. They consist of a set V of vertices and a set E of *edges*. Edges are (in contrast to arcs) defined as *unordered pairs* of vertices, that is, we define them by $E \subseteq \{\{u, v\} \mid u, v \in V\}$. Here, the edges $\{u, v\}$ and $\{v, u\}$ are equal, however, the definitions of an edge's tail and head do not apply. For the remainder of this section we define notion using directed graphs. Most definitions carry over to undirected graphs, naturally.

Given a graph $G = (V, A)$, the *reverse graph* $\overleftarrow{G} = (G, \overleftarrow{A})$ is obtained from G by flipping all arcs, i. e., it holds that $(u, v) \in \overleftarrow{A} \Leftrightarrow (v, u) \in A$. (Note that for the

undirected case $G = \overleftrightarrow{G}$.)

A *vertex-induced subgraph* (sometimes we may just write *subgraph* for short) of G, is a graph $G' = (V', A')$, such that $V' \subseteq V$ holds true, and A' contains exactly the arcs from A for which both incident vertices are in V'. More precisely, $A' = \{(u, v) \mid (u, v) \in A \text{ and } u, v \in V'\}$. A less common type of subgraph is the *arc-induced subgraph* $G' = (V', A')$. Here the set of arcs is defined as $A' \subseteq A$ and the set of vertices is induced from A' by $V' = \{u \mid u \in V \text{ and } \exists (u, v) \in A' \text{ or } (v, u) \in A'\}$.

Attributes, Costs, and Functions. A graph $G = (V, A)$ can be augmented by assigning further attributes to its vertices and/or arcs. Of particular interest in this work are arc costs which model the criterion one likes to optimize (for example, travel time). Generally speaking, a *cost function* $\ell \colon A \to \mathbb{R}$ maps each arc to a real number. If $a = (u, v) \in A$ is an arc, we interchangeably write $\ell(a)$ and $\ell(u, v)$ to refer to its cost. Note that often costs are nonnegative, i. e., $\ell \geq 0$. For example, negative values for lengths or travel times make little sense.

Section 4.3 shows how timetables can be modeled as a graph. Because vehicles in a timetable operate at specific times only, we require the notion of *time-dependent cost* (as opposed to the previously defined *time-independent* or *constant* cost). More precisely, $\ell \colon A \to \mathbb{F}$ now maps each arc to a *function* f from a *function space* $\mathbb{F}$. We refrain from using negative values from now on, hence, functions $f \in \mathbb{F}$ are of the form $f \colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$. For the purposes of this work, it is important that domain and codomain of f have the same "unit". In our case of travel times, f maps *departure time* to *travel time* (which have both the unit *seconds*). This allows us to define binary *link* (composition) and *merge* operations on $\mathbb{F}$. Given two functions $f, g \in \mathbb{F}$, the link operation is defined as

$$\operatorname{link}(f, g) := f + g \circ (f + \mathrm{id}). \tag{3.1}$$

Here, id denotes the identity function, i. e., $\mathrm{id}(x) = x$ for any value x, and "$\circ$" denotes the function composition operator. Sometimes we use the (equivalent) infix notation $f \oplus g$ to refer to $\operatorname{link}(f, g)$. Note that the link operation is neither commutative nor associative, which makes the order of evaluation important. For *constant* functions, i. e., $f \equiv \ell_1$ and $g \equiv \ell_2$, the link operation simplifies to $\operatorname{link}(f, g) = \ell_1 + \ell_2$.

For two functions $f, g \in \mathbb{F}$, the merge (sometimes we also call it minimum) operation is, on the other hand, defined as

$$\operatorname{merge}(f, g) = \min(f, g). \tag{3.2}$$

Here, by min we denote the component-wise minimum of two functions, that is, $\min(f, g)(x) = \min(f(x), g(x))$ for any value x. Note that $\mathbb{F}$ is closed under both merge and link operations. Finally, by $\underline{f}$ we denote the *minimum* value $\min(f(x))$ of f and with $\bar{f}$ the *maximum* value $\max(f(x))$ for any $x \in \mathbb{R}_{\geq 0}$.

If for a given value $\Pi \in \mathbb{R}_{\geq 0}$ it holds that $f(x) = f(x \mod \Pi)$, we call f *periodic* with *period* Π. Moreover, we say that $f \in \mathbb{F}$ fulfills the *FIFO-property*, if and only if

$$x_1 \leq x_2 \;\Rightarrow\; x_1 + f(x_1) \leq x_2 + f(x_2) \quad \text{for any } x_1, x_2 \in \mathbb{R}_{\geq 0} \tag{3.3}$$

holds true. Sometimes we also refer to this property as *non-overtaking property*: If x_1, x_2 are departure times mapped to travel times by f, Equation (3.3) basically states that departing later (at x_2) will never make one arrive earlier (i. e., *before* $x_1 + f(x_1)$). If any $f \in \mathbb{F}$ fulfills the FIFO-property, we simply say that the whole function space $\mathbb{F}$ fulfills it. We now show that the FIFO-property is preserved by the merge and link operations.

Lemma 1. *Let $f, g \in \mathbb{F}$ be functions that fulfill the FIFO-property, then the functions obtained by* $\text{link}(f,g)$ *and* $\text{merge}(f,g)$ *also fulfill the FIFO-property.*

Proof. We first prove the lemma for the link operation. Because f and g fulfill the FIFO-property, we have for any $x_1, x_2 \in \mathbb{R}_{\geq 0}$:

$$\begin{aligned} x_1 \leq x_2 \;\Rightarrow\; & x_1 + f(x_1) \leq x_2 + f(x_2) \quad \text{and} \\ & x_1 + g(x_1) \leq x_2 + g(x_2). \end{aligned} \tag{3.4}$$

Recall that for any $x \in \mathbb{R}_{\geq 0}$, link is defined as $\text{link}(f,g)(x) = x + g(x + f(x))$. Thus, we obtain:

$$\begin{aligned} x_1 \leq x_2 \;&\Rightarrow\; x_1 + g(x_1) \leq x_2 + g(x_2) && \text{(3.5a)} \\ &\Leftrightarrow\; x_1 + x_1 + g(x_1) \leq x_2 + x_2 + g(x_2) && \text{(3.5b)} \\ &\Leftrightarrow\; x_1 + x_1 + g(x_1 + f(x_1)) \leq x_2 + x_2 + g(x_2 + f(x_2)) && \text{(3.5c)} \\ &\Leftrightarrow\; x_1 + \text{link}(f,g)(x_1) \leq x_2 + \text{link}(f,g)(x_2). && \text{(3.5d)} \end{aligned}$$

Note that Equation (3.5b) follows from $x_1 \leq x_2$ and Equation (3.5c) holds because of the FIFO-property of f. We now consider the merge operation. Again, because the FIFO-property holds for f and g individually, we obtain from Equation (3.4):

$$\begin{aligned} x_1 \leq x_2 \;\Rightarrow\; & \min[x_1 + f(x_1), x_1 + g(x_1)] \leq x_2 + f(x_2) \quad \text{and} && \text{(3.6a)} \\ & \min[x_1 + f(x_1), x_1 + g(x_1)] \leq x_2 + g(x_2), && \text{(3.6b)} \end{aligned}$$

from which immediately follows:

$$\begin{aligned} x_1 \leq x_2 \;&\Rightarrow\; \min[x_1 + f(x_1), x_1 + g(x_1)] \leq \min[x_2 + f(x_2), x_2 + g(x_2)] && \text{(3.7a)} \\ &\Leftrightarrow\; x_1 + \min[f(x_1), g(x_1)] \leq x_2 + \min[f(x_2), g(x_2)] && \text{(3.7b)} \\ &\Leftrightarrow\; x_1 + \text{merge}(f,g)(x_1) \leq x_2 + \text{merge}(f,g)(x_2). && \text{(3.7c)} \end{aligned}$$

This exactly proves our claim. We conclude that a function space $\mathbb{F}$ is even closed under merge and link, if $\mathbb{F}$ fulfills the FIFO-property. ∎

Paths, Cycles, and Trees. Of central relevance to this work is the notion of paths, cycles, and trees, which we introduce next. Given a (directed) graph $G = (V, A)$, a *path* P is a sequence of vertices $P = [u_1, \dots, u_k]$ such that for every subsequent pair of vertices $u_i u_{i+1}$, the arc (u_i, u_{i+1}) is contained in A. If u_1 and u_k coincide, we call P a *cycle*. A *subpath* P' of P, also written $P' \subseteq P$, is a subsequence of P's vertices.

Two vertices u, v are *connected* in G, if there exists a u–v-path in G. If this is true for all pairs of vertices, we call G *strongly connected*. A connected subgraph G' of G is called *strongly connected component* (of G). A graph $G = (V, A)$ is called *tree rooted* at $u \in V$, if $|A| = |V| - 1$ holds, and for every vertex $v \in V$ exists a u–v-path in G.

The *cost* of a path P is the sum of the arc's costs along P. More precisely,

$$\ell(P) := f(v_1, v_2) \oplus f(v_2, v_3) \oplus \cdots \oplus f(v_{k-1}, v_k), \tag{3.8}$$

which simplifies to $\sum_{i=1}^{k-1} \ell(v_i, v_{i+1})$, if all arcs have constant cost. (Recall that $\oplus$ is the infix notation of the link operation.) In the nonconstant case, the result $\ell(P)$ is a function itself, i. e., $\ell(P) \in \mathbb{F}$, hence, we may sometimes also write f_P for $\ell(P)$.

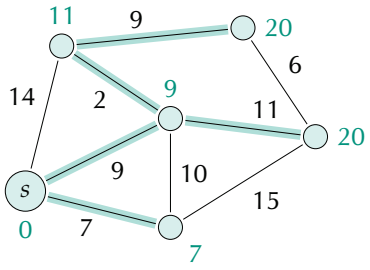


Figure 3.1. A (constant) weighted graph with shortest path tree rooted at vertex s.

For designated *source* and *target* vertices $s, t \in V$, an s–t-path $P_{s,t}$ is a path such that $u_1 = s$ and $u_k = t$. Let from now on the arc's costs be constant. This allows us to define the notion of shortest paths: For given vertices s and t, a *shortest s–t-path* is an s–t-path with minimum cost (among all s–t-paths that exist in G). Let $P_{s,t}$ be a shortest s–t-path, then the cost $\ell(P_{s,t})$ is also referred to as *distance* (from s to t), denoted by $\operatorname{dist}(s, t)$. If no (shortest) path from s to t exists in G, we define $\operatorname{dist}(s, t) = \infty$. Moreover we define $\operatorname{dist}(u, u) = 0$ for any vertex $u \in V$. In the case shortest paths are unique (that is, for every pair $s, t \in V$ there is at most *one* shortest path from s to t), the union of all shortest paths originating at a common source vertex s forms a tree, called *shortest path tree* (rooted at s). Figure 3.1 shows an example of a (weighted) graph with a shortest path tree rooted at vertex s. Thick green edges are part of the tree (representing shortest paths), and the green label of a vertex u denotes its distance from s, i. e., $\operatorname{dist}(s, u)$.

Metrics. A *metric* is a function $d \colon \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ on the reals, that satisfies the following properties for any values $x, y, z \in \mathbb{R}$:

$$\begin{aligned}
&d(x, y) \geq 0 && \text{(non-negativity)}\\
&d(x, y) = 0 \text{ if } x = y && \text{(identity)}\\
&d(x, y) = d(y, x) && \text{(symmetry)}\\
&d(x, z) \leq d(x, y) + d(y, z). && \text{(triangle inequality)}
\end{aligned}$$

If all properties but symmetry hold for d, the function d is called a *quasimetric*. Note that, if all arc costs are non-negative, the distance function dist (as defined above)

in graphs is a quasimetric, as can be easily checked: Non-negativity and identity hold by definition. Regarding the triangle inequality, assume that it is false. This implies vertices u, v, w, such that $\text{dist}(u, w) > \text{dist}(u, v) + \text{dist}(v, w)$. However, the concatenation of the corresponding shortest paths $P_{u,v}$ and $P_{v,w}$ results in an u–w-path P with cost $\ell(P) = \text{dist}(u, v) + \text{dist}(v, w) < \text{dist}(u, w)$ which is a contradiction. If the graph is undirected, dist is a full metric, since for any vertices $u, v \in V$ reversing the shortest u–v-path always yields a shortest v–u-path.

3.2. Partitions

Some of the algorithms in this work use partitions. While partitions can be defined over any set of entities, we introduce them in the context of graphs.

Given a graph $G = (V, A)$, a *partition* of the vertices V is a family $\mathcal{C} = \{C_1, \ldots, C_k\}$ of *cells* $C_i \subseteq V$, such that each vertex $u \in V$ is contained in exactly one cell C_i. We augment this definition to multiple levels, as follows. A *nested multilevel partition* of L levels is a family $\{\mathcal{C}^1, \ldots, \mathcal{C}^L\}$ of partitions with nested cells, that is, for each level $\ell \leq L$ and cell $C_i^\ell \in \mathcal{C}^\ell$ there must exist a cell $C_j^{\ell+1} \in \mathcal{C}^{\ell+1}$ on level $\ell + 1$, such that $C_i^\ell \subseteq C_j^{\ell+1}$ holds. We call $C_j^{\ell+1}$ the *supercell* of C_i^ℓ and C_i^ℓ a *subcell* of $C_j^{\ell+1}$. For consistency, we define $\mathcal{C}^0 = V$ and $\mathcal{C}^{L+1} = \{V\}$. In other words, $\mathcal{C}^0$ consists of a singleton cell for each vertex, while $\mathcal{C}^{L+1}$ consists of a single cell that contains all vertices. An arc $(u, v) \in A$ is called a *boundary arc* on level ℓ, if and only if u and v are in different cells of $\mathcal{C}^\ell$. In this case, u and v are called *boundary vertices* (of level ℓ). Note that a boundary vertex of level ℓ is also a boundary vertex on all lower levels. The union of all boundary arcs for a given level ℓ is called the *cut* of level ℓ. Usually, we aim for partitions with a small cut. Figure 3.2 shows an example of a two-level partition of a graph. Boundary vertices in the figure are marked black, while inner vertices are shallow.

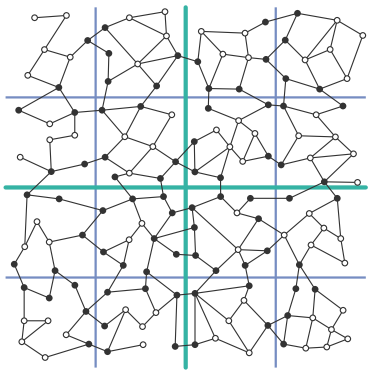


Figure 3.2. A graph partitioned into two nested levels.

3.3. Regular Languages and Finite Automata

Relevant to multimodal route planning are (regular) languages and finite automata. They are used to define admissible sequences of transportation modes. We, therefore, formally introduce basic notion for (regular) languages and finite automata in the following.

Languages. All languages are based on symbols. Also formal languages are based on this entity. We call a finite set Σ of symbols *alphabet*. A sequence of symbols $w =$

$[\sigma_1, \sigma_2, \ldots, \sigma_k]$ from Σ is called *word*. Often, we omit the square brackets and just write $w = \sigma_1\sigma_2 \cdots \sigma_k$ for short. The *empty symbol* (sometimes also called *empty word*) is denoted by ε and has zero length. Moreover, it holds for any word w that $w\varepsilon = w$ and $\varepsilon w = w$. The *length* of a word, denoted by $|w|$ is the number of (non-empty) symbols it is composed of. For two words $w_1 = \sigma_1 \cdots \sigma_k$ and $w_2 = \sigma_{k+1} \cdots \sigma_l$, the *concatenation* of w_1 with w_2, denoted w_1w_2 is defined as $w_1w_2 = \sigma_1 \cdots \sigma_k\sigma_{k+1} \cdots \sigma_l$.

A (not necessarily finite) set of words L over Σ is called a *langauge*. Regular set operations, such as union and intersection, also apply to languages. For a given language L, the i-th *power language* is defined, as follows. For $i = 0$, we set $L^0 := \{\varepsilon\}$. For any $i > 0$, we define L^i recursively by $L^i := \{ww' \mid w \in L^{i-1} \text{ and } w' \in L\}$. Having this notion at hand, the *Kleene closure* of a language L is defined as

$$L^* = \bigcup_{i \geq 0} L^i. \tag{3.10}$$

Note that if L is not empty, then L^* always yields a set with an infinite (but countable) number of words. For the special case of $L = \Sigma$, the set Σ^* contains all words that can be formed by symbols from Σ. Note that the empty word is always part of L^*. Finally, given two languages L_1 and L_2, the *concatenated language* $L_1 \cdot L_2$ is obtained by $L_1 \cdot L_2 := \{ww' \mid w \in L_1 \text{ and } w' \in L_2\}$.

Of particular interest for multimodal route planning are *regular languages*. They are a special class of languages according to the following definition.

Definition 1. *Given an alphabet Σ, then a language L over Σ is called regular if and only if it can be constructed by the following recursive rules.*

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

Note that set intersection is excluded explicitly as a construction rule.

Automata. Having established the notion of regular languages, we now introduce another representation of them, namely finite automata. A *nondeterministic finite automaton* $\mathcal{A}$ is a tuple $\mathcal{A} = (S, \Sigma, \delta, I, F)$ that consists of a set S of *states*, a set Σ of *symbols* (again, also called *alphabet*), a *transition function* $\delta\colon S \times \Sigma \to \mathcal{P}(S)$ (note that $\mathcal{P}(S)$ denotes the *power set* of S) which maps a state and symbol to a set of states; a set I of *initial states*, and a set F of *final states*. Note that any nondeterministic finite automaton can be converted into a *deterministic* one. For a deterministic automaton it has to hold that $|I| = 1$, and the transition function maps to S instead of $\mathcal{P}(S)$. In this work, however, we focus on nondeterministic automata, since they suit our purpose best.

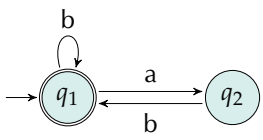


Figure 3.3. A finite automaton.

Often, we define automata in terms of their *transition graph* $G_{\mathcal{A}} = (V, A)$. Here, vertices from V depict states (in other words, $V = S$), and there is an arc $uv \in A$, if and only if there exists a symbol $\sigma \in \Sigma$ such that $v \in \delta(u, \sigma)$ holds. In this case, we label uv with σ in $G_{\mathcal{A}}$. In this work, we mark initial states by an incoming arrowp-tip, while final states are twin-framed. Figure 3.3 shows an example of a finite automaton that has two states over the alphabet $\Sigma = \{a, b\}$ and accepts the language $L = (\{b\} \cup \{ab\})^*$, that is, all words that may contain any numbers of a and b, however, any a must be followed by a b. Note that the automaton is deterministic.

To connect finite automata to regular languages, we say that, given a (regular) language L, a word $w \in L$ is *accepted* by a finite automaton $\mathcal{A}$, if there is a path P in the transition graph $G_{\mathcal{A}}$ of $\mathcal{A}$ that starts with at an initial state from I, ends with a final state from F, and where the subsequent arcs of P are labeled by the subsequent symbols of w. If no such path exists, the word w is *rejected* by $\mathcal{A}$. If every word from L is accepted by $\mathcal{A}$, then $\mathcal{A}$ accepts the language L. Kleene's Theorem (see [Kle56, RS59] for details) states that regular languages and finite automata are equivalent: For every regular language L there exists a (nondeterministic) finite automaton $\mathcal{A}$, such that a word w is accepted by $\mathcal{A}$ if and only if it is in L. On the other hand, given a finite automaton $\mathcal{A}$, the words accepted by $\mathcal{A}$, always form a regular language. We, therefore, use the terms regular language and finite automaton interchangeably in this work.

Chapter 4

Public Transit Journey Planning

This chapter is devoted to journey planning in public transit networks. In order to develop algorithms that compute journeys, we will first carefully define the underlying input of a public transit network in a mathematical sense. In this work, we consider schedule-based networks, i. e., the networks are specified in terms of their timetable. The timetable includes all stops of the network and also vehicles which operate at predefined times of the day along certain sequences of stops, as well as, footpaths that enable transfers between nearby stops.

Problems. Going from there, we introduce the problems we are considering in this chapter. In all of them, we are given origin and destination stops p_s and p_t, and are then interested in computing "optimal" journeys from p_s to p_t. While, for example, in (static) road networks the definition of an "optimal" journey is relatively straight-forward (e. g., the journey that minimizes travel time), this is not necessarily true for public transit networks. This is due to two reasons: time-dependency and multiple important criteria. Regarding the former, transit networks are inherently time-dependent, that is, the optimal journey depends on the departure time. To that extent, we consider the earliest arrival problem, where the departure time τ is given as an additional input. Here, a journey is optimal if it arrives at the destination stop p_t as early as possible while not departing before τ at the origin stop p_s. If, instead of a departure time τ, we are given a whole time range Δ as input, the goal is to compute *all* optimal (e. g., with respect to travel time) journeys from p_s to p_t that depart within Δ. This type is called range problem (or profile problem, if Δ specifies the whole operational time period of the timetable). Note that for time-dependent road networks these two types of problems have been studied as well [DW09b].

Secondly, just optimizing for a single criterion (such as arrival time) may not be enough. Usually other criteria, such as the number of transfers, or (monetary) cost, are just as important. We tackle this by computing Pareto sets of journeys that

minimize each criterion independently. The Pareto set contains all journeys which do not dominate each other, that is, for no two journeys one is better in all considered criteria than the other. Multicriteria optimization augments both the earliest arrival and the range problem.

Models. Since the shortest path problem is well understood in the literature, a common approach to computing journeys takes the timetable as input and builds a graph from it, such that shortest paths correspond to optimal journeys. There exist several different graph models that represent the timetable, who can roughly be partitioned into two classes: Time-expanded and time-dependent models. The former "expands" time in the sense that it contains a vertex for every event in the timetable (such as a particular vehicle departing at a certain stop). Unfortunately, this yields graphs of large size. Therefore, the time-dependent model aims to compress the graph in the sense that it condenses vehicles that operate on the same segment of the network into a single arc. The cost of an arc is then no longer constant, but depends on the time of day (hence, the name of the model).

We first recap the widely used time-expanded and time-dependent graph models. We then improve the time-dependent approach by modeling conflicting vehicles inside stops more carefully. The key idea is to compute a (minimum) coloring of a corresponding conflict graph, such that each color represents a vertex in the model graph. Hence, using this *Coloring Model*, we are able to reduce the size of the graph significantly, which directly accelerates any graph search algorithm running on it.

Moreover, for realistic queries *footpaths* are crucial to enable transfers between stops. However, often such data is not available from the input. Thus, we present a heuristic approach to generate artificial footpaths using the underlying road network. Our method is based on snapping stops to (nearest) intersections and introducing cliques between stops of the same intersection.

Algorithmic Approaches. Having set up the graph models, we describe basic algorithmic approaches that solve the earliest arrival, range, and multicriteria problems. In particular, we describe Dijkstra's algorithm, which can be easily adapted to both the time-expanded and time-dependent graph models. It is also the basis of all the other algorithms for the more enhanced problems.

Starting from there, we introduce our two main contributions of this chapter: A new algorithm that computes range (and profile) queries efficiently, and a new approach to solve multicriteria earliest arrival and multicriteria range queries. Both algorithms compute in their basic variant optimal journeys from a source stop p_s to *all* other stops of the network, but can be accelerated if one is only interested in journeys to a designated target stop p_t.

Parallel Self-Pruning Connection Setting Algorithms. The key idea of the first algorithm, called Self-Pruning Connection Setting (SPCS), is that the number of possible journeys is bounded by the number of outgoing connections from the source stop p_s. Moreover, all time-dependent travel time distances in public transit networks networks can be described by piecewise linear functions that have a representation bounded by this number as well. Also, only few connections prove useful when traveling sufficiently far away. The algorithm we present greatly exploits this fact by pruning such connections as early as possible. To this extent, we introduce the notion of *connection-setting*, that can be seen as an extension of the label-setting property of Dijkstra's algorithm, which usually is lost in profile searches. Unlike previous algorithms, which are notoriously hard to parallelize (see [MBBC09] and [MS03]), we parallelize SPCS (which we then call PSPCS) in a multicore scenario by distributing different connections outgoing from p_s to different CPU cores. Furthermore, we show how connections can be pruned even across different cores.

While one-to-all queries are relevant for the preprocessing of many speedup techniques (see, e. g., [DW09b, DPW09a]), we also accelerate the more common scenario of point-to-point queries explicitly. Therefore, we propose to utilize the very same algorithm for valuable preprocessing. The key idea is that we select a small number of important stops (called *hub stops*) and precompute a full distance table between all these stops, which then can be used to prune the search during the query.

Round-Based Public Transit Optimized Router. The second algorithm is RAPTOR, our Round-bAsed Public Transit Optimized Router. It considers multicriteria optimization and computes all Pareto-optimal journeys—minimizing the arrival time and the number of transfers made. Unlike the previously mentioned approaches, RAPTOR is *not* Dijkstra-based. Instead, it operates in rounds, one per transfer, and computes arrival times by traversing every *route* (such as a bus line) at most once per round. The algorithm boils down to a dynamic program with simple data structures and excellent memory locality. RAPTOR can also be parallelized in a multicore scenario by distributing independent routes among multiple CPU cores.

We also introduce two extensions of RAPTOR. The first, McRAPTOR, generalizes RAPTOR to handle more criteria, beyond arrival time and transfers. As examples we use fare zones, a common pricing model, and the reliability of transfers. The second extension we propose, rRAPTOR, computes bicriteria range queries, which output full Pareto sets of journeys for all departures within a time range. Because our algorithms do not rely on preprocessing, they are fully dynamic, easily handling delays, trip cancellations, or route changes.

Overview. Section 4.1 formally defines timetables, which are the basis of our public transit networks. Section 4.2 then introduces the problems we are interested to solve in this work. Section 4.3 revisits existing approaches for modeling timetables as

graphs and introduces our new Coloring Model. Existing algorithmic approaches to compute journeys are recapped in Section 4.4. Section 4.5 then introduces our new approach to compute range and profile queries, called Self-Pruning Connection Setting Algorithm (SPCS). Section 4.6 introduces our new Round-bAsed Public Transit Optimized Router (RAPTOR) that computes multicriteria journeys.

4.1. Inputs

In this section we define the input to our route planning problems. We start by giving a formal introduction to *timetables*, which form the basis of a public transit network. We define the timetable in a "natural" way using the notion of *stops*, *routes*, and *trips*. For some algorithms, however, a different view, using the notion of *elementary connections* is more useful, which we derive next. Finally, we also define the *output* of our problems. In our case of route planning in public transit networks, these are usually (sets of) journeys.

Timetables. Our algorithms work on a *timetable* $\mathfrak{T} = (\Pi, \mathcal{S}, \mathcal{T}, \mathcal{R}, \mathcal{F})$ where $\Pi \subset \mathbb{Z}_{\geq 0}$ is the *period of operation* (think of it as the seconds of a day), $\mathcal{S}$ is a set of *stops*, $\mathcal{T}$ a set of *trips*, $\mathcal{R}$ a set of *routes*, and $\mathcal{F}$ a set of *footpaths* (sometimes also called *transfers*).

Elements $\tau \in \Pi$ are called *time points*. Each stop in $\mathcal{S}$ corresponds to a distinct location in the network where one can board or get off a vehicle (bus, tram, train, etc.). Typical examples are bus stops and train platforms. Each trip $t \in \mathcal{T}$ represents a sequence of stops a specific vehicle (train, bus, subway, etc.) visits along a line. At each stop in the sequence, it may drop off or pick up passengers. Moreover, each stop p in a trip t has associated arrival and departure times $\tau_{\text{arr}}(t, p), \tau_{\text{dep}}(t, p) \in \Pi$, with $\tau_{\text{arr}}(t, p) \leq \tau_{\text{dep}}(t, p)$. The first and last stops of a trip have undefined arrival and departure times, respectively. The trips in $\mathcal{T}$ are partitioned into routes: Each route in $\mathcal{R}$ consists of the trips that share the same sequence of stops. Also, we require the trips within a route to be non-overtaking (i. e., no trip overtakes any other within the same route). Typically, there are many more trips than routes. Footpaths in $\mathcal{F}$ model walking connections (or transfers) between stops. Each footpath consists of two stops p_1 and p_2 with an associated constant walking time $\ell(p_1, p_2)$. Sometimes, we require $\mathcal{F}$ to be transitive: If p_1 and p_2 are indirectly connected by footpaths, (p_1, p_2) is contained in $\mathcal{F}$ as well. The length $\ell(p_1, p_2)$ then depicts the minimum time to get from p_1 to p_2 using a sequence of footpaths. Finally, a stop $p \in \mathcal{S}$ has an associated *minimum change time* $\tau_{\text{ch}}(p)$, the minimum time required to change trips at p (due to long walking distances within p, for example). Note that the minimum change time can be zero for some stops.

Sometimes we require to measure the *duration* between two time points $\tau_1, \tau_2 \in \Pi$. We therefore use a *difference function* δ, which simply evaluates to $\delta(\tau_1, \tau_2) = \tau_2 - \tau_1$. In the case we consider periodic timetables, δ is computed by $\tau_2 - \tau_1$ if $\tau_2 \geq \tau_1$

Table 4.1. Exemplary excerpt of typical input data from the London timetable of 2011. Each row represents one elementary connection.

Route and Trip No.	Idx.	Departure Stop	Dep.-Time	Arrival Stop	Arr.-Time
			...		
Bakerloo-0	2	Charing Cross	06:46	Piccadilly Circus	06:48
Bakerloo-0	3	Piccadilly Circus	06:48	Oxford Circus	06:50
Bakerloo-0	4	Oxford Circus	06:50	Regent's Park	06:52
Bakerloo-0	5	Regent's Park	06:52	Baker Street	06:54
Bakerloo-0	6	Baker Street	06:54	Marylebone	06:56
			...		
Victoria-6	1	Green Park	15:21	Oxford Circus	15:22
Victoria-6	2	Oxford Circus	15:22	Warren Street	15:24
Victoria-6	3	Warren Street	15:24	Euston	15:26
Victoria-6	4	Euston	15:26	King's Cross St. Pancras	15:26
			...		

and $\Pi + \tau_2 - \tau_1$ otherwise. Note that δ is not symmetric.

Elementary Connections. Given a timetable $\mathfrak{T}$, we may derive a set $\mathcal{C}$ of *elementary connections*. Intuitively speaking, elementary connections are the smallest entity into which a timetable can be decomposed. We require them for defining the graph models, as well as for the algorithms. More formally, an *elementary connection* $c \in \mathcal{C}$ is a tuple $c = (t, p_{\text{dep}}, p_{\text{arr}}, \tau_{\text{dep}}, \tau_{\text{arr}})$, which is interpreted as trip $t \in \mathcal{T}$ going from stop $p_{\text{dep}} \in \mathcal{S}$ to stop $p_{\text{arr}} \in \mathcal{S}$, departing at p_{dep} at time $\tau_{\text{dep}} \in \Pi$ and arriving at $\tau_{\text{arr}} \in \Pi$. For simplicity, given an elementary connection c, the function $X(c)$ selects the X-entry of c. For example, $\tau_{\text{dep}}(c)$ refers to the departure time of c. Table 4.1 shows an exemplary excerpt from the set of elementary connections for the timetable of London (an input we often use). For any connection c, the column "Route" refers to the associated route of c, and "Index" depicts the ordinal sequence number of c along its route. The table shows partial trips for two subway (tube) routes, namely of the Bakerloo line and the Victoria line.

Journeys. Any journey-planning algorithm operating on a timetable outputs a set of *journeys* $\mathcal{J}$. A journey is defined as a sequence of trips and footpaths in the order of travel. In addition, each trip in the sequence is associated with two stops, corresponding to the pick-up and drop-off points. Note that a journey containing k trips has exactly $k - 1$ transfers. Journeys are associated with several optimization criteria. We say a journey J_1 *dominates* a journey J_2, denoted by $J_1 \preccurlyeq J_2$, if J_1 is no worse in any criterion than J_2. A set of pairwise nondominating journeys is a *Pareto set*. In our algorithms we use *labels* (often associated with stops) for intermediate

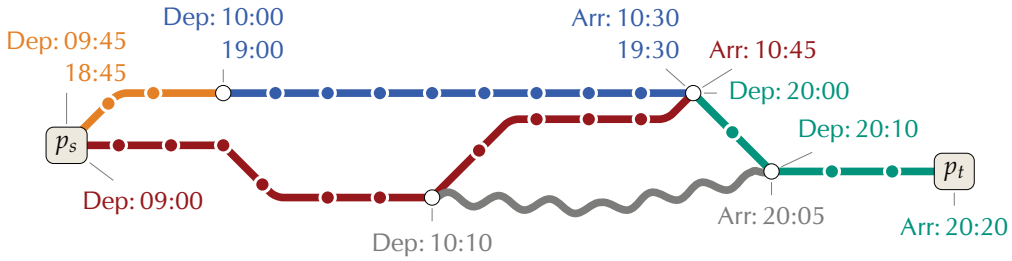


Figure 4.1. Example of three journeys from p_s to p_t. The departure time is set to τ = 09:00. Annotations depict departure/arrival times of trips on the route of respective color. The snaky line illustrates a very long route in the network.

journeys. The definition of domination translates to labels naturally.

4.2. Problems

In this section we formally define the problems that we consider in this chapter.

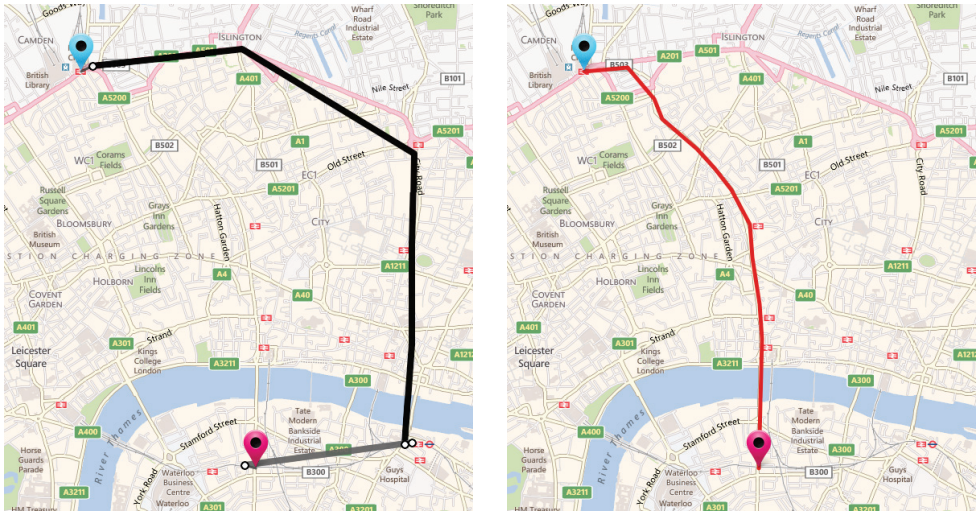
4.2.1. Earliest Arrival Problem

The simplest problem we are considering is the *earliest arrival problem*. Given a public transit timetable, a source stop p_s, a target stop p_t, and a departure time τ, it asks for a journey that departs at p_s no earlier than τ and arrives at p_t as *early as possible*. An algorithm which solves the earliest arrival problem is also called *earliest arrival query*.

The solution of the earliest arrival problem consists of (at most) one journey, namely the one which arrives at p_t earliest. We call this journey *optimal*. Often, more than one optimal journey exists, in which case we break ties arbitrarily. On the other hand, if no journey matching the requested criteria exists, the output is just the empty set.

Tight Journeys. Unfortunately, computing the earliest arrival solution does not necessarily output the journey with minimum *travel time*. This may seem counterintuitive at first[1], however, imagine a low-frequency bus route which must be taken as the last leg of *any* journey in order to reach the target p_t. If the departure time τ is chosen such as there is sufficient "slack" time until the first (feasible) trip departs toward p_t, all journeys that somehow "spend" this slack time by going around the transit network are optimal. See Figure 4.1 for an example. It depicts three journeys from p_s to p_t for a departure time of 9:00. The earliest trip for the last leg of the journey arrives at p_t at 20:20. All three illustrated journeys are optimal (i. e., they

[1] In time-independent networks, e. g., static road networks, computing earliest arrival and minimum travel time journeys is equivalent.



(a) Arrival time: 11:08; one transfer.

(b) Arrival time: 11:09; zero transfers.

Figure 4.2. Exemplary solution to the multicriteria problem for a query from *King's Cross St. Pancras* station to *Southwark* station in London at 10:50. Optimization criteria are arrival time and the number of transfers taken. The left solution uses the Northern and Jubilee tube lines, while the right solution uses bus line 63.

share the same earliest arrival time of 20:20). In particular, the snaky gray trip is part of a very long route where the slack time can be spent.

To remedy this issue, we extend the earliest arrival problem to the *tight earliest arrival problem* as follows. Given a public transit timetable, a source stop p_s, a target stop p_t, and a departure time τ, it asks for a journey that departs at p_s no earlier than τ and arrives at p_t as early as possible. From all such journeys, it further asks for the one that departs from p_s *latest*. This results in a journey that is "tight" regarding the arrival time in the sense that there is no other journey with a smaller travel time for the considered departure time. Note that, in general, the solution is still not necessarily unique, in which case we break ties arbitrarily. In Figure 4.1 taking the orange route at 18:45, the blue one at 19:00, and, finally, the green one at 20:00 results in a tight earliest arrival journey for the departure time of 9:00 at p_s.

4.2.2. Multicriteria Problem

The multicriteria problem is a generalization of the earliest arrival problem taking more than one optimization criterion into account. However, in this chapter, we always require arrival time to be part of the criteria. Examples for further criteria include the number of transfers taken or the monetary cost of a journey. Formally, in the *multicriteria problem* one is given a public transit timetable, a source stop p_s, a

target stop p_t, and a departure time τ. It then asks for a (full) *Pareto sets* of journeys $\mathcal{J}$, for which the following must hold. Each journey $J \in \mathcal{J}$ must not leave p_s earlier than τ, and for any two journeys $J_1, J_2 \in \mathcal{J}$ neither J_1 may dominate J_2, nor J_2 may dominate J_1. On the other hand, for any journey J from p_s to p_t that departs after τ and is *not* included in $\mathcal{J}$, there must be a witness journey $J' \in \mathcal{J}$, such that J' dominates J. An algorithm which solves the multicriteria problem is called a *multicriteria query*. An example is shown in Figure 4.2. It shows two journeys, one which arrives one minute earlier than the other, but having one more transfer.

4.2.3. Range Problem

The range problem no longer requires a specific departure time as input, but rather takes a *time range* (as the name implies), for which optimal journeys are computed. More precisely, given a public transit timetable, a source stop p_s, a target stop p_t, and a time range $\Delta \subseteq \Pi$ (recall that Π is the timetable's period), the *range problem* asks for a *minimal* set of journeys, such that for each departure time $\tau \in \Delta$ exists a journey $J_\tau \in \mathcal{J}$ that departs at p_s no earlier than τ and arrives at p_t as early as possible. Note that requiring a minimal set of journeys implies that if two journeys J_1 and J_2 with the same arrival time exist in $\mathcal{J}$, only the one with later departure time from p_s is kept. If the input range Δ equals the full period Π of the timetable, the problem is also called *profile problem*. An algorithm which solves the range or profile problem is called *range* or *profile query*.

Note that the range problem can be interpreted as a special case of the multicriteria problem in the following sense. It takes a time range Δ instead of a departure time τ as input and considers two criteria: arrival time and *departure time*. It then computes a Pareto set $\mathcal{J}$ of journeys, such that any journey $J \in \mathcal{J}$ departs within Δ, and for any two journeys J_1 and J_2, the journey J_1 dominates J_2 if and only if J_1 departs no earlier and arrives no later than J_2. If additional criteria (besides arrival and departure time) are considered, we also call the problem *multicriteria range problem*.

4.2.4. Reverse Problems

Up to now, all problems are specified in terms of their departure times at the source stop p_s. If, instead, one is interested to optimize for a given *arrival time* at the target stop p_t, any of the previous problems can be reversed. For the case of the earliest arrival problem, we obtain the *latest departure counterproblem*. It takes a source stop p_s, a target stop p_t, and an arrival time τ as input, and it asks for a journey J from p_s to p_t that arrives at p_t no *later* than τ and departs at p_s as late as possible. The range and multicriteria problems are defined analogously. Note that the notion of tight journeys also carries over: In addition to asking for a journey that departs from p_s as late as possible, we also require it to arrive at p_t as *early* as possible (while not arriving later than τ).

Usually, the reverse problems are equivalent to their forward counterparts: An algorithm that computes the forward problem can be used for the respective reverse problem by *inverting* the input: Stop sequences of all routes are reversed (i. e., they are now operated in reverse order). Then, for every trip and every stop the departure and arrival times are swapped. Also, all times occurring in the timetable are translated by mirroring them at a sufficiently high value (e. g., $\Pi + 1$). By these means, time is considered in reverse order, and an algorithm that minimizes arrival time corresponds to maximizing departure time on the original input. Hence, for the rest of this work, we only focus on the forward problems.

4.3. Graph Models

This section presents several graph-based models that build a directed graph $G = (V, A)$ from the timetable. The idea is to model the graph in such a way that problems from Section 4.2 can be solved by (possibly augmented) shortest path algorithms. Recall that the timetable is inherently time-dependent (vehicles operate at well-defined times during the day). Therefore, the graph must capture the notion of *time-dependency* to yield meaningful solutions. Two distinct approaches exist: The *time-expanded approach* (Section 4.3.2) expands time in the sense that for every event of the timetable a vertex is cerated. The *time-dependent approach* 4.3.3 combines trips of the same route into one arc, significantly reducing the graph size. Another (much simpler) model is the *stop model* (Section 4.3.1). It is not useful to compute queries, however, often used as a preprocessing ingredient to speedup techniques.

Note that using a graph to model the timetable is very common in the literature. Time-expanded models have been first used (in the context of public transit) in [SWW00, SWZ02], while time-dependent models have been first used in [Nac95, BJ04, PSWZ04b]. A more recent overview of the different graph-based modeling approaches is also available in [PSWZ08].

Contributions and References. New contributions in this section are the (time-dependent) Coloring Model (Section 4.3.4), which significantly reduces the graph size of the time-dependent model for earliest arrival queries, as well as a heuristic that generates artificial footpaths (Section 4.3.5). These are crucial for computing realistic journeys. Both of these sections are based on [DKP12], which appeared in the ACM Journal of Experimental Algorithmics, vol. 17, no. 1 in 2012. It is joint work with Bastian Katz and Daniel Delling.

4.3.1. Stop Model

The simplest model that represents the timetable is the *stop model*. It was first introduced in [SWZ02], where it was called *station graph*. It builds a directed

graph $G = (V, A)$ where each vertex from V exactly corresponds to a *stop* $p \in \mathcal{S}$ of the timetable. We, therefore, refer to vertices by their stops and just write $p \in V$ for short. Arcs are then inserted as follow. An arc $p_i p_j$ from vertex p_i to p_j is contained in A if and only if there exists an elementary connection that goes from p_i to p_j. More formally, there exists an elementary connection $c \in \mathcal{C}$ for which $p_{\text{dep}}(c) = p_i$ and $p_{\text{arr}}(c) = p_j$ hold. The cost $\ell(p_i, p_j)$ of an arc $p_i p_j$ is the minimum travel time of all elementary connections from p_i to p_j, i. e.,

$$\ell(p_i, p_j) = \min\{\delta(\tau_{\text{dep}}(c), \tau_{\text{arr}}(c)) \mid c \in \mathcal{C} \text{ and } p_{\text{dep}}(c) = p_i \text{ and } p_{\text{arr}}(c) = p_j\}. \quad (4.1)$$

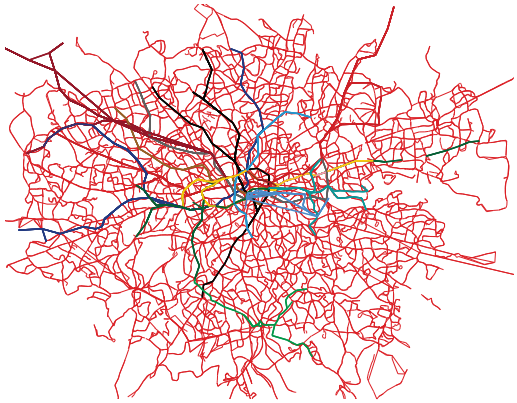


Figure 4.3. Stop graph of greater London.

Figure 4.3 shows the stop graph for the timetable of the greater London area. Arcs between stops served by bus routes (the vast majority) are drawn thin and red, while all other arcs (Tube, DLR, and ferry boats) are drawn with respect to their official route color from the London network map.

While the stop model is certainly very simple, it does not capture the time-dependent nature of the timetable. Because arc costs are defined in terms of minimum travel times between stops, shortest paths only correspond to *lower bounds* on the actual (total) travel time. In fact, none of the problems from Section 4.2 can be solved by this model. However, it is useful as a preprocessing ingredient for some speedup techniques, such as ALT [DPW09b], Arc-Flags [DPW09b, BGM10, BDGM09], or SHARC [Del11]. In Section 4.5.3 we use it to compute "importance" values for stops. We use the most important stops to compute a full distance table which then helps accelerating earliest arrival queries.

4.3.2. Time-Expanded Model

The time-expanded approach remedies the issues of the stop model by encoding time-dependencies into the graph via the notion of *events*. There exist two basic variants of this approach, the *simple* and the *realistic* model [PSWZ08]. They are described in the following. Finally, we explain, how footpaths can be integrated into the model.

Simple Model. Given a timetable $\mathfrak{T}$ and its set $\mathcal{C}$ of elementary connections, the simple time-expanded model basically inserts two vertices *per connection*, which are interconnected by an arc. More formally, it defines the *simple time expanded model graph* $G = (V, A)$. For every connection $c \in \mathcal{C}$ it creates two vertices: A *departure*

vertex $u_{\text{dep}}(c)$ and an *arrival vertex* $u_{\text{arr}}(c)$. Vertices in the time-expanded approach always have (implicit) associated *timestamps* $\tau(u)$. In our case the timestamp of a departure vertex $u_{\text{dep}}(c)$ is given by the departure time of the respective connection $\tau_{\text{dep}}(c)$, while the timestamp of an arrival vertex $u_{\text{arr}}(c)$ is given by the arrival time $\tau_{\text{arr}}(c)$. Analogously, each vertex has an (implicit) associated *stop* value $p(u)$.

Arcs are created as follows. For every connection $c \in \mathcal{C}$ the model inserts an arc $(u_{\text{dep}}(c), u_{\text{arr}}(c)$ between the connection's departure and arrival vertices. By these subsequent connections of the same trip become interconnected by arcs. To allow changing of trips inside stops, the model inserts, independently for each stop $p \in \mathcal{S}$, *transfer arcs* uv between subsequent vertices u and v of p in order of increasing time. More precisely, there is a transfer arc $uv \in A$ if and only if $p(u) = p(v)$, $\tau(u) \leq \tau(v)$ and there is no other vertex w with $p(w) = p(u)$ and $\tau(u) < \tau(w) < \tau(v)$. All arcs uv are weighted by the time difference of their respective incident vertices, i. e., $\ell(u, v) = \delta(\tau(u), \tau(v))$.

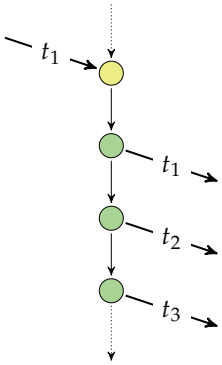


Figure 4.4. Simple time-expanded model.

If multiple vertices with the same timestamp exist for a stop p, the model may *merge* them into a single vertex. By these means, and under the assumption that no connection has a duration of zero, all arcs point in direction of increasing time. Thus, the simple time-expanded graph G is acyclic, i. e, it does not contain any cycles. For the case that the timetable is periodic, the model adds, at each stop p an additional arc uv from the *latest* to the *earliest* vertex at p, enabling transfers from one period of the timetable to the next. Note that by these means the graph is no longer acyclic.

Figure 4.4 shows an example of four connections belonging to three trips t_1, t_2, and t_3 at some stop p in the simple time-expanded model graph. Arrival vertices are filled yellow, while departure vertices are filled green. Vertices are drawn such that time increases from top to bottom.

As opposed to the stop model (cf. Section 4.3.1), it can be easily seen that any path in the time-expanded graph correspond to a valid journey for the (input) timetable. Another interesting observation is that (for the aperiodic case) in fact *any* path P between two vertices s and t is also a *shortest s–t-path*. Recall that arc costs are defined in terms of the time difference of their incident vertices. Therefore, the cost of the individual arcs of P must exactly sum up to $\delta(\tau(s), \tau(t))$. A major disadvantage of the simple time-expanded model is, however, that arbitrary quick transfers at stops are possible, since it does not incorporate the minimum change times defined in the timetable.

Realistic Model. The realistic time-expanded model extends the simple model by incorporating minimum change times at stops.

Formally, it defines a directed graph $G = (V, A)$ as follows. Similarly to before, it creates for each connection $c \in \mathcal{C}$ a departure vertex $u_{\text{dep}}(c)$, an arrival vertex $u_{\text{arr}}(c)$,

and adds the arc $u_{\text{dep}}(c)u_{\text{arr}}(c)$ with cost $(\tau(u_{\text{dep}}(c), \tau(u_{\text{arr}}(c)))$ to A. To enable minimum change times, the model additionally creates, for each connection, a *transfer vertex* $u_{\text{tr}}(c)$, which it assigns to stop $p_{\text{dep}}(c)$—i. e., $p(u_{\text{tr}}(c)) = p_{\text{dep}}(c)$. Analogously, the timestamp of $u_{\text{tr}}(c)$ is set to $\tau(u_{\text{dep}}(c))$, which is equivalent to the departure time of c. Besides the already mentioned *connection arcs*, the model adds additional *transfer arcs* into the model. For each connection it connects the transfer vertex $u_{\text{tr}}(c)$ to the departure vertex $u_{\text{dep}}(c)$ with cost zero. Moreover, to enable staying within a trip t, the model adds, for each connection $c \in \mathcal{C}$ of trip t an arc from the arrival vertex $u_{\text{arr}}(c')$ of the *preceding* connection c' of t to the departure vertex $u_{\text{dep}}(c)$ of c. For the first connection of t no preceding connection exists and no arc is created. Note that $u_{\text{arr}}(c')$ and $u_{\text{dep}}(c)$ must belong to the same stop by definition.

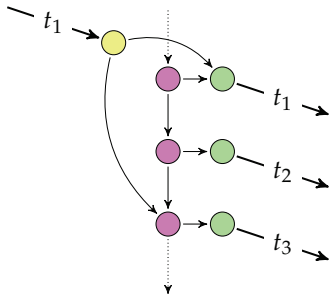


Figure 4.5. Realistic time-expanded model.

To enable transfers within a stop $p \in \mathcal{S}$, additional transfer arcs are created: For each arrival vertex u at p, the model determines the first transfer vertex v (also at p), for which $\tau(u) + \tau_{\text{ch}}(p) \leq \tau(v)$ holds true, and it adds an arc between these vertices, accordingly. If no such vertex v exists, no arc is added. Moreover, subsequent transfer vertices are interconnected by arcs in increasing order of their timestamp (similarly to the simple model). Transfer vertices with the same timestamp may, again, be merged.

An example of the realistic time-expanded model graph is shown in Figure 4.5. It depicts the same trips and connections as Figure 4.4. Arrival vertices are filled yellow, departure vertices are filled green, and transfer vertices are filled purple. Note that transferring from trip t_1 to t_2 is not possible due to the minimum change time (however, continuing the journey in trip t_1 is very well possible). This could not be captured by the simple time-expanded model of Figure 4.4.

Footpaths. Footpaths are integrated into the model as follows. For every existing footpath $(p_i, p_j) \in \mathcal{F}$ with length $\ell(p_i, p_j)$, it adds several arcs between vertices of p_i and p_j. More precisely, it adds from every arrival vertex u at p_i an arc to the *earliest* transfer vertex v at p_j for which $\tau(u) + \ell(u, v) \leq \tau(v)$ holds true. (Note that, footpaths in the simple model are added in a similar manner between arrival and *departure* vertices.)

Discussion. The main advantage of the time-expanded approach is that the resulting graph is time-independent, i. e., all arcs have constant cost. This enables simple queries algorithms: Essential, Dijkstra's algorithm [Dij59] can be applied out of the box to compute journeys. Moreover, for the case that the timetable is aperiodic, the resulting graph is acyclic, which enables even simpler query algorithms, such as Connection Scan [DPSW13].

On the downside, the size of the time-expanded graphs is rather huge, since every

event of the timetable is modeled by at least one vertex. In fact, the number of vertices and arcs are both in the order of $\mathcal{O}(|\mathcal{C}|)$. Additionally, modeling footpaths is somewhat more complicated: Even though a footpath (p_i, p_j) is already time-independent in the original data, it still creates as many arcs in G as there are arrival events at the stop p_i.

Some extensions to the time-expanded approach exist that were omitted in this section. These include incorporating traffic days and variable transfer times at stops [PSWZ08]. Also, further engineering the model helps accelerating query performance [DPW09b].

4.3.3. Time-Dependent Model

In contrast to the time-expanded approach, the time-dependent model aims for smaller graphs whose number of vertices and arcs is roughly in the order of the number of stops and routes of the timetable. Instead of vertices that correspond to events of the timetable, time-dependency is encoded as a special form of *time-dependent travel time functions* on the arcs—hence, the name of the model.

We first describe how the travel time functions look like and discuss how they can be efficiently linked and merged (cf. Section 3.1). We then recap the simple and realistic time-dependent graph models [PSWZ08] and, finally, explain how footpaths are integrated into the model.

Travel Time Functions. Recall that in the time-expanded graph (cf. Section 4.3.2), each elementary connection $c \in \mathcal{C}$ is modeled by an arc $uv \in A$ with constant weight $\delta(\tau_{\text{dep}}(c), \tau_{\text{arr}}(c))$. The key idea of the time-dependent approach is to *combine* several elementary connections into a *single* arc by a *time-dependent travel time function*. We, therefore, consider a *function space* $\mathbb{F}$ consisting of *travel time functions* of the form $f\colon \Pi \to \mathbb{Z}_{\geq 0}$. Each function $f \in \mathbb{F}$ maps a *departure time* onto a *travel time* (or cost). Departure times are taken from the interval Π, which is the timetable's period of operation, while travel times may assume arbitrary nonnegative integers (think of a train arriving after midnight).

The travel time functions in our scenario must encode elementary connections that operate at specific times with respect to the timetable. Hence, each elementary connection $c \in \mathcal{C}$ that is represented by the function f, creates a *connection point* $q_c = (\tau_{\text{dep}}(c), \delta(\tau_{\text{dep}}(c), \tau_{\text{arr}}(c))$, such that evaluating f at departure time $\tau = \tau_{\text{dep}}(c)$ results in the respective travel time $f(\tau) = \delta(\tau_{\text{dep}}(c), \tau_{\text{arr}}(c))$. For simplicity we write $\tau_{\text{dep}}(q)$ and $\tau_{\text{tra}}(q)$ to refer to the departure and travel times encoded by the connection point q. Now let P_f be the *set of connection points* of f. Values of f between subsequent connection points are obtained by interpolation via waiting. More precisely, let $\tau \in \Pi$ be an arbitrary departure time and q be the "next" connection point, i. e., the connection point for which $\delta(\tau, \tau_{\text{dep}}(q))$ is minimal. Then, f is evaluates

at τ to

$$f(\tau) = \underbrace{\tau_{\text{tra}}(q)}_{\text{Travel time associated with } q} + \overbrace{\delta(\tau, \tau_{\text{dep}}(q))}^{\text{Waiting time for } q\text{'s departure}}. \tag{4.2}$$

Note that if the connection points of f are kept sorted by their departure times, evaluating f takes time $\mathcal{O}(\log|P_f|)$ by using binary search, and time $\mathcal{O}(|P_f|)$ using a simple linear scan. Preliminary experiments indicated that the algorithmic overhead of a binary search results in worse practical performance than using a linear scan. (Note that a linear scan admits excellent spatial locality and is, hence, extremely cache-friendly.) In this work, we additionally use the following interpolation heuristic when scanning P_f. Let τ be the departure time at which f is evaluated, then we check the connection point q at index $i = \tau/\Pi \cdot |P|$. If $\tau \geq \tau_{\text{dep}}(q)$, we continue scan P_f in ascending order, otherwise, we scan in descending order.

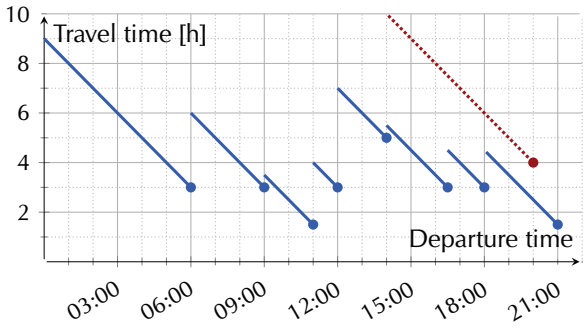


Figure 4.6. Piecewise linear travel time function.

To ensure correctness of the model, the travel time functions must fulfill the FIFO-property (cf. Section 3.1). In the context of connection points, this is interpreted as follows: There must be no two subsequent connection points $q_1, q_2 \in P$ such that $\delta(\tau_{\text{dep}}(q_1), \tau_{\text{dep}}(q_2)) + \tau_{\text{tra}}(q_2) \leq \tau_{\text{tra}}(q_1)$ holds true. In other words, skipping q_1 and waiting for q_2 must not result in a smaller overall travel time. Note that simply deleting q_1 restores the FIFO-property of f.

Figure 4.6 illustrates an exemplary travel time function f with eight connection points. Connection points are indicated by dots and line segments indicate interpolation by waiting. The red connection point violates the FIFO-property and, hence, must not be included in f.

Given two functions f_1 and f_2, the *link operation* can be efficiently implemented by a linear sweep algorithm. For every connection point q_1 from f_1, it looks for the connection point q_2 from f_2 that minimizes $d = \delta(\tau_{\text{dep}}(q_1) + \tau_{\text{tra}}(q_1), \tau_{\text{dep}}(q_2))$ and inserts the connection point $q = (\tau_{\text{dep}}(q_1), d + \tau_{\text{tra}}(q_2))$ into the resulting function's connection point set. If no such q_2 exists, no new connection point is created. Likewise, no connection point q is created, if q would violate the FIFO-property. Note that the number of connection points in the output function is bounded by $\min\{|P_{f_1}|, |P_{f_2}|\}$ and the link algorithm runs in time $\mathcal{O}(|P_{f_1}| + |P_{f_2}|)$.

Finally, the *merge operation* of two functions f_1 and f_2 can also be implemented by a linear scan. The resulting function simply consists of the union $P_{f_1} \cup P_{f_2}$ of each function's connection points, discarding those who violate the FIFO-property. As a

result, the number of connection points in the output is bounded by $|P_{f_1}| + |P_{f_2}|$ and the merge operation also runs in time $\mathcal{O}(|P_{f_1}| + |P_{f_2}|)$.

Simple Model. After having set up the notion of travel time functions, we now describe the *simple time dependent model* [BJ04]. Given a timetable $\mathfrak{T}$ and its corresponding set of elementary connections $\mathcal{C}$, the model builds a directed graph $G = (V, A)$ by creating one vertex u_p for each stop $p \in \mathcal{S}$. For simplicity, we write p and u_p interchangeably. Arcs are created as follows. The model inserts the arc p_1p_2 into A if and only if there exists an elementary connection $c \in \mathcal{C}$ that goes from p_1 to p_2, i. e, for which $p_{\text{dep}}(c) = p_1$ and $p_{\text{arr}}(c) = p_2$ holds. Note that up to this point the model exactly matches the stop model (cf. Section 4.3.1).

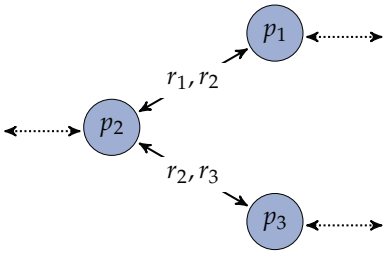


Figure 4.7. Simple time-dependent model.

To make the model time-dependent, arc costs are defined in terms of travel-time functions, that is, $\ell\colon A \to \mathbb{F}$. Each arc $p_1p_2 \in A$, thereby, contains exactly those connection points that correspond to elementary connections that travel from p_1 to p_2. Connection points that violate the FIFO-property may either be discarded or put on a separate parallel arc (in case they may not be omitted). Note that if we evaluate the lower bound $\underline{f}_{p_1p_2}$ of the travel time function at each arc p_1p_2, we exactly obtain the stop model (cf. Section 4.3.1). Figure 4.7 illustrates the simple time-dependent model on three stops p_1, p_2, and p_3.

Realistic Model. Like in the time-expanded scenario, the simple time-dependent model fails to capture minimum change times at stops. The realistic model aims to remedy this issue by slightly blowing up the graph [PSWZ04b, PSWZ08]. Instead of a single vertex per stop, multiple vertices are created. The model is thereby based on the intuition, that changing between trips of the same *route* is never optimal. Therefore, the model groups elementary connections by their route. More precisely, let $\mathcal{R}_p$ be the set of routes that serve stop $p \in \mathcal{S}$. (We may sometimes refer to them by stop-routes.) The model now still creates a *stop vertex* $p \in V$ (like before), but additionally creates a *route vertex* r_p for every stop-route from $\mathcal{R}_p$.

Arcs are created as follows. For each route $r \in \mathcal{R}$ of the timetable, and two subsequent stops $p_i, p_j \in \mathcal{S}$ that are served by the route r, the model creates a time-dependent *route arc* $r_{p_i}r_{p_j} \in A$ whose travel time function contains a connection point for every elementary connection $c \in \mathcal{C}$ for which $p_{\text{dep}}(c) = p_i$, $p_{\text{arr}}(c) = p_j$ and $r(c) = r$ hold. Again, non-FIFO connection points are either discarded or put on separate parallel arcs. To enable transfers between trips of different routes, the model additionally creates *transfer arcs* that connect the stop vertex to (and from) all corresponding route vertices. More formally, it adds arcs p, r_p and r_p, p for every

stop-route $r_p \in \mathcal{R}_p$. Note that these arcs have *constant* cost. More precisely, the model charges cost for the minimum change time by setting $\ell(pr_p) = \tau_{\text{ch}}(p)$ (recall that $\tau_{\text{ch}}(p)$ depicts the minimum change time at stop p) for each arc that goes from the stop vertex to the route vertices. Accordingly, it sets $\ell(r_p p) = 0$ for arcs from route vertices to the stop vertex.

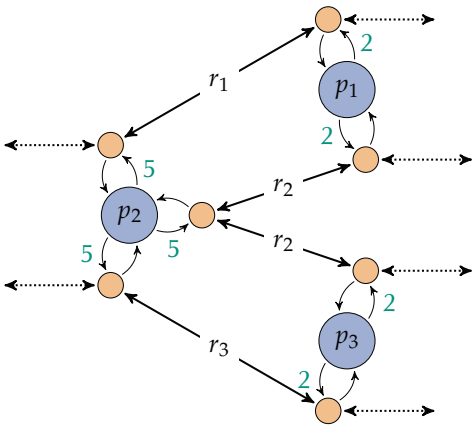


Figure 4.8. Realistic time-dependent model.

Figure 4.8 shows the same stops and routes in the realistic model as Figure 4.7 showed them for the simple model. Route vertices are drawn smaller in orange. Note that changing routes at stop p_2 now requires five minutes time as it is indicated by the green labels on the transfer arcs.

Footpaths. To incorporate footpaths between stops, the graph is augmented by arcs between stop vertices. Recall that each footpath of the input is defined as a tuple $(p_i, p_j) \in \mathcal{S} \times \mathcal{S}$ with associated length $\ell(p_i, p_j)$, meaning that it is possible to walk from stop p_i to stop p_j in time $\ell(p_i, p_j)$. To incorporate them, we insert, for each tuple $(p_i, p_j) \in \mathcal{F}$ an arc (p_i, p_j) into G with constant weight $\ell(p_i, p_j)$—similarly to transfer arcs *within* stops.

Discussion. Like for the time-expanded model, several extensions exist that were omitted here. These include incorporating traffic days and enabling variable transfer times between routes [PSWZ08]. However, all of the (time-dependent) models share the notion of time-dependent arcs in order to combine elementary connections into a single arc. By these means, the graphs obtained by these models are significantly smaller when compared to the time-expanded approach. This comes at the cost of a (slightly) more complicated query algorithm, though, which must evaluate travel time functions when considering arcs. However, in practice this is greatly outweighed by the smaller graph sizes, making the time-dependent approach the more practical one [BDW11].

Still, all (realistic) variants of the time-dependent model rely on the notion of routes and add at least as many vertices per stop to the graph as there are routes serving it. In fact, an analysis of the model reveals that the average number of route vertices per stop is typically between 5 and 16, depending on the input (cf. Section 4.5.4), which is quite high. To reduce this number, the next section introduces a new model which is based on a formal notion of *conflicting* trips. Note that a smaller graph size immediately results in faster query times for *any* search algorithm.

4.3.4. Coloring Model

One main reason of using the notion of routes in the realistic time-dependent model is the observation that in any journey, transfers between two trips of the same route are never beneficial. Thus, when assigning trips of the same route to the same route vertex (i. e., assigning their respective elementary connections to arcs incident to the route vertex), we ensure that we do not generate a journey with invalid transfers, i. e., violating the minimum change time at some stop. However, this property can also be guaranteed by a more formal notion of *conflicting trips*.

Now, consider two trips t_1 and t_2 which serve some stop p. Let $\tau_{\text{arr}}(t_1, p)$ be the arrival time of trip t_1 at p and $\tau_{\text{dep}}(t_2, p)$ the departure time of t_2 at p. Then, these two trips *conflict* if and only if t_2 departs after the arrival of t_1 and the time in between is smaller than the minimum change time at t. More precisely, t_1 and t_2 conflict if and only if

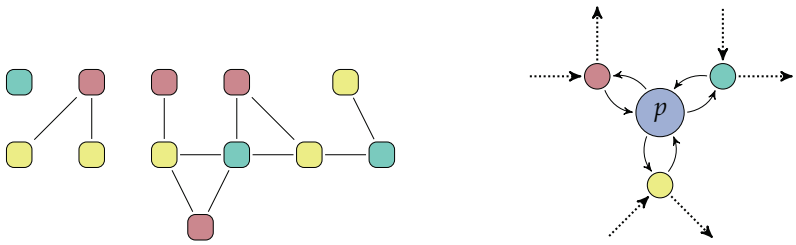
$$\tau_{\text{dep}}(t_2, p) \geq \tau_{\text{arr}}(t_1, p) \quad \text{and} \quad \tau_{\text{arr}}(t_1, p) + \tau_{\text{ch}}(p) > \tau_{\text{dep}}(t_2, p). \tag{4.3}$$

In this case, putting t_1 and t_2 on the same route vertex could produce an illegal journey, which must be avoided.

Testing the conflict condition for all pairs of trips serving p naturally induces an undirected conflict graph $G^*(p) = (V^*(p), E^*(p))$. The vertex set $V^*(p) \subseteq \mathcal{T}$ contains exactly those trips $t \in \mathcal{T}$ that serve p (i. e., where there exists an elementary connection $c \in \mathcal{C}$ with $t(c) = t$ and $p_{\text{dep}}(c) = p$ or $p_{\text{arr}}(c) = p$. Two pairs of vertices $t_i, t_j \in V^*(p)$ are connected by an edge $\{t_i, t_j\} \in E^*(p)$ if and only if t_i and t_j are conflicting. Experiments on our instances (cf. Section 4.5.4) reveal that the number of conflicting trips is small indeed: We observe that of all possible trip pairs per stop, on average less than 0.5 % are actually conflicting. Thus, we may regard G^* as sparse.

It is now easy to see that a vertex coloring of $G^*(p)$ (i. e., each vertex gets a color assigned), where no two adjacent vertices may share the same color, induces a set of route vertices of the stop p in the model graph G: Let K be the number of distinct colors used for $G^*(p)$, then for each color $k = 1 \dots K$ we create a route vertex u in G and put exactly those trips onto u that have assigned color k in $G^*(p)$. An example of a conflict graph and its induced subgraph in the time-dependent model is illustrated in Figure 4.9.

Computing Colorings. In general, our goal is to generate as few route vertices in G as possible. Thus, we aim for computing a coloring on $G^*(p)$ with as few colors as possible. In fact, a lower bound on the number of route vertices for p in p is given by the chromatic number $\chi(G^*(G))$. Since it is well known that computing $\chi(G^*(p))$ is NP-complete [Kar72], we use the following greedy heuristic to color $G^*(p)$ for every p independently. We start with an uncolored graph and process the vertices of $G^*(p)$ in order of decreasing degree. When considering vertex u, we assign u the smallest color that is not used by any of u's neighbors.



(a) Conflict graph. (b) Model graph.

Figure 4.9. Exemplary conflict graph $G^*(p)$ of a stop p with a valid vertex coloring that uses three colors (left) and the corresponding induced subgraph for p of the time-dependent model having three route vertices (right). In the right figure, route arcs are drawn bold while transfer arcs are drawn thinner.

Note that this algorithm never uses more than $\text{maxdeg}(G^*(p)) + 1$ colors, where $\text{maxdeg}(G^*(p))$ depicts is the maximum vertex degree of $G^*(p)$. Since we consider $G^*(p)$ to be sparse, the results of the greedy algorithm on $G^*(p)$ are quite good in practice (see Section 4.5.4 for experimental details).

Merging Small Stops. To further reduce the number of vertices in the time-dependent model graph G, we may merge small stop p which have only one route vertex (i. e., $G^*(p)$ has been colored with one color). More precisely, we merge the stop vertex with the (only) route vertex. Since there are no conflicting trips at p, we do not lose correctness by applying this procedure to all stops of this type in G.

4.3.5. Artificial Footpaths

Considering footpaths turns out crucial for finding realistic journeys with reasonable transfers. Even worse, the graph obtained from real-world timetables may even get disconnected into several components when footpaths are omitted. Unfortunately, footpath data is not always included with the available timetable data from the transit agencies. Thus, we propose the following heuristic to generate an artificial set $\mathcal{F}$ of footpaths.

Let $\mathfrak{R}$ be the road network covering (at least) the geographical area of the public transit network for which we are about to generate footpaths. Our heuristic then assigns every stop $p \in \mathcal{S}$ to a bucket b using $\mathfrak{R}$. Each intersection of the road network maintains a bucket. The algorithm then finds for stop p the intersection $b \in \mathfrak{R}$ which is *geographically closest* to p, and assigns p to b if the geographical distance is no greater than a parameter (typically set to 100 m). It then looks at all buckets b it created and adds, between all pairs of different stops $p_i, p_j \in b$, a footpath (p_i, p_j) to $\mathcal{F}$. The length of (p_i, p_j) is obtained by the sum of the distances from p_i to b and from b to p_j divided by an assumed average walking speed (typically 4 kph).

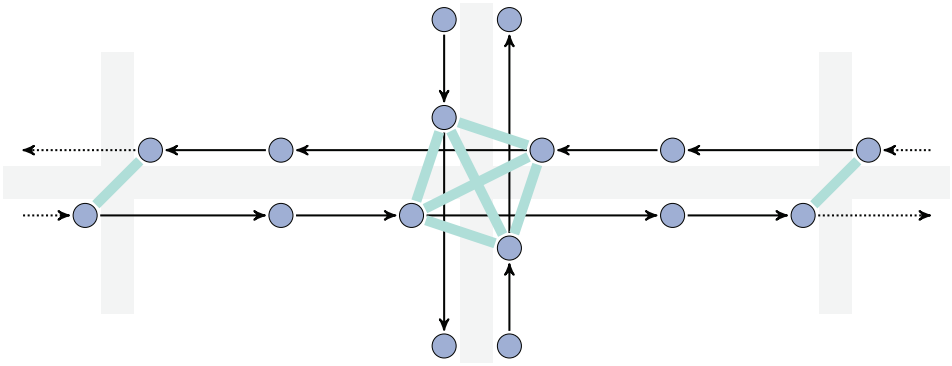


Figure 4.10. Example of heuristically generated footpaths in a typical U. S. bus network (two bus lines along two streets). Footpaths are depicted as highlighted arcs.

Note that since each stop is assigned to exactly one bucket, our heuristic obtains many small components of stops that are interconnected by footpaths near intersections. In particular, we avoid connecting large regions of the network through sequences of footpaths. See Figure 4.10 for an example.

4.4. Basic Algorithms

In this section we describe basic algorithmic approaches for solving the problems from Section 4.2. They all operate on one of the models we introduced in Section 4.3 and are variants of extensions of Dijkstra's algorithm [Dij59]. Moreover, we also use them as benchmark to evaluate the performance of our new algorithms in Sections 4.5 and 4.6.

We group the algorithms in this section by the problems they solve. Therefore, we start in Section 4.4.1 with the simplest problem, the earliest arrival problem and show how Dijkstra's algorithm can be adapted to the time-expanded [SWW00] and time-dependent [CH66] models. Then, in Section 4.4.2 we consider the multicriteria and range problems. We first describe two algorithms which can be applied to both problems: The label-correcting (LC) algorithm [Dea99] extends Dijkstra's algorithm by propagating collections of labels (bags) through the network, while the multi-label-correcting (MLC) algorithm [PSWZ08, DMS08] still maintains a bag of labels at each vertex, but propagates them individually. Both algorithms share the property, that vertices may be scanned more than once. Finally, we also recap the Layered Dijkstra (LD) [BJ04] algorithm. It is simpler in that it does not use bags and can be applied to the bi-criteria problem where the second optimization criterion (besides arrival time) is discrete and assumes a small number of different values (e. g., number of transfers).

4.4.1. Earliest Arrival Problem

Recall from Section 4.2 that for the earliest arrival problem we are given source and target stops $p_s, p_t \in \mathcal{S}$ and a departure time τ. We are now interested in computing a journey that departs p_s no earlier than τ and arrives at p_t as early as possible. This problem can be solved by Dijkstra's algorithm [Dij59] on both the time-expanded and time-dependent graph models (on the latter it requires one minor modification, which we discuss later).

Dijkstra's Algorithm. Given a directed graph $G = (V, A)$ with source and target vertices $s, t \in V$ (we discuss mapping p_s, p_t to s, t later), Dijkstra's algorithm maintains two data structures: A *priority queue* Q of vertices as well as *arrival time labels* $\tau(u)$ for every vertex $u \in V$, initialized in the beginning to infinity. It starts by setting $\tau(s) = \tau$ and adding s to Q with key τ. It now, in turn, *extracts* (or *scans*) the vertex u with minimum (current) key from the priority queue. It then proceeds by *scanning* all outgoing arcs $a = (u, v) \in A$ (in any order). For each, it creates a tentative arrival time label $\tau_{\text{tent}}(v) = \tau(u) + \ell(a)$ at v. If $\tau_{\text{tent}}(v)$ improves $\tau(v)$, i. e., $\tau_{\text{tent}}(v) < \tau(v)$ holds, it *relaxes* a: It updates $\tau(v)$ to $\tau_{\text{tent}}(v)$ and updates Q to contain v with key $\tau_{\text{tent}}(v)$. The algorithm stops when Q runs empty.

Note that up to now, Dijkstra's algorithm computes arrival times for *all* vertices of G. We observe that it scans vertices in the order of increasing arrival time from s, since it always extracts the vertex with minimum key. Thus, if we are only interested in the arrival time for a *single* vertex t (point-to-point query), it may stop, as soon as t is scanned. At this time, the target vertex t is guaranteed to have the correct arrival time set. Note that this approach can be generalized to computing distances to a *set* of target vertices.

Running Time. The running time of Dijkstra's algorithm is determined by the data structure used for the priority queue Q. Every vertex is scanned at most once, resulting in $|V|$ extractions from Q. Also, every arc is scanned at most once, which results in up to $|A|$ updates of Q. Using a binary heap, yields a running time of $\mathcal{O}((|V| + |A|) \log |V|)$ [CLRS01]. This can be improved by, e. g., Fibonacci Heaps to $\mathcal{O}(|A| + |V| \log |V|)$ [FT87] or, if all arc costs are integral in the range $[0, C]$, Multi-Level Bucket Queues to $\mathcal{O}(|A| + |V| \sqrt{\log C})$ [DF79].

Throughout this work we use binary heaps in our experiments for two reasons. First, our graphs are sparse, i. e., $|A| \in \mathcal{O}(|V|)$, thus, the theoretical running time reduces to $\mathcal{O}(|V| \log |V|)$, and, second, their implementation is simple, thus, admitting good spatial (memory) locality, which helps cache performance in practice.

Source and Target Vertices. It remains to discuss how we select s and t for the stops p_s and p_t. In the time-expanded model, the source vertex s is selected, among all *departure* vertices (simple model), respective all *transfer* vertices (realistic model),

to be the one with minimum timestamp $\tau(s)$ that is greater than τ. Moreover, the departure time, which is used to initialize the algorithm at s, is updated to $\tau(s)$.

Unfortunately, the target vertex t is unknown before the query is executed. (Note that knowing t immediately yields an (earliest) arrival time $\tau(t)$.) Still, Dijkstra's algorithm may stop, as soon as *any* vertex u of p_t has been scanned. The first scanned vertex at p_t provably corresponds to the earliest arrival time at p_t. We call the resulting algorithm *Time-Expanded Dijkstra* (TED).

In the time-dependent models, on the other hand, p_s and p_t are directly mapped to the corresponding stop vertices, which are then used as source and target vertices of Dijkstra's algorithm.

Time-Dependency. Dijktra's algorithm can be adapted to handle time-dependent arc costs quite easily [CH66]. The algorithm remains essentially the same, except that when it scans an arc $a = (u, v)$, the tentative label $\tau_{\text{tent}}(v)$ is computed by evaluating the time-dependent arc function f_a at time $\tau(u)$. (Recall that $\tau(u)$ corresponds to the earliest arrival time at u.) We call the resulting algorithm *Time-Dijkstra* (TD). Pseudocode of TD is shown in Figure 4.11.

Aperiodic Timetables. In case the input timetable is aperiodic, the time-expanded model enables a simpler algorithm to compute earliest arrival queries. It is based on the observation that, in this case, the resulting (both simple and realistic) time-expanded graphs are *acyclic*, i. e., they do not contain cycles. Therefore, *any* path between two vertices $u, v \in V$ is also a *shortest* path. To see why, recall that vertices have associated timestamps, and every arc's cost exactly corresponds to the time difference of its incident vertices. Hence, the earliest arrival problem can be reduced to a *reachability problem*: Given a source vertex $s \in V$, determine the smallest (in terms of its timestamp) reachable vertex u at the target stop p_t. This vertex can be computed by, e. g., breadth-first search (from s), which runs in time $\mathcal{O}(|V| + |A|)$. A more sophisticated approach, called Connection Scan Algorithm (CSA), orders the arcs topologically in a preprocessing step and scans them by a linear sweep during the query [DPSW13].

Timestamps. If many queries are run on the same graph subsequently, a significant amount of time in Dijkstra's algorithm is spent resetting all labels to infinity during the initialization phase. The algorithm may avoid this by keeping a global *clock* ω, initially set to zero. Moreover, every vertex maintains a timestamp (also initially set to zero). Instead of setting all labels to infinity in the beginning, the algorithm just increases the clock value ω by one. Then, each time it updates a vertex label, it also sets its corresponding timestamp to ω. Whenever it attempts to read a label, it first checks if its timestamp equals the current clock value ω. If not, the label's value

```
// Input: Graph G = (V, A), source vertex s, target vertex t, departure time τ
// Side Effects: Earliest arrival times τ(u) at all vertices u ∈ V, if t = ⊥ or at t, otherwise
// Initialization of the algorithm
1  Q ← new PQueue()                                   // Create empty priority queue
2  τ(·) ← ∞                                           // Initialize arrival time labels
3  τ(s) ← τ
4  Q.Insert(s, τ)
// Main loop
5  while not Q.Empty() do
6      u ← Q.ExtractMin()                             // Scan next vertex
7      if u = t then                                  // Stopping criterion
8          stop;
9      forall the outgoing arcs a = (u, v) ∈ A do     // Scan outgoing arcs
10         τ_tent(v) ← τ(u) + f_a(τ(u))               // Compute tentative arrival time at v
11         if τ_tent(v) < τ(v) then                   // Improve arrival time at v?
12             τ(v) ← τ_tent(v)                       // Update label of v
13             if not Q.Contains(v) then              // Update priority queue
14                 Q.Insert(v, τ_tent(v))
15             else
16                 Q.DecreaseKey(v, τ_tent(v))
```

Figure 4.11. Time-Dependent Dijkstra (TD).

does not stem from the current execution. Hence, it is discarded and assumed to be infinity, instead.

4.4.2. Multicriteria and Range Problems

Recall that the multicriteria and range problems have in common that they may output more than one journey. To reflect this, any algorithm that computes such queries must maintain a (dynamic) *collection* of labels per vertex (instead of a single label). In the following, we quickly recap three algorithms that are based on Dijkstra's algorithm and can be applied to, both, multicriteria and range queries. We consider the Label-Correcting algorithm (LC), the Multi-Label-Correcting algorithm (MLC), and the Layered Dijkstra algorithm (LD).

Label-Correcting Algorithm. The Label-Correcting Algorithm (LC) [Dea99] extends Dijkstra's algorithm by maintaining a collection of labels $B(u)$ at each vertex $u \in V$, called *bag*. For the scenario of multicriteria queries, every label $L \in B(u)$ has an

associated value per optimization criterion (arrival time always being among them). The algorithm maintains the invariant that $B(u)$ is a Pareto set at every vertex u, i. e., no two labels $L_1, L_2 \in B(u)$ *dominate* each other. (Recall that L_1 dominates L_2, denoted $L_1 \preccurlyeq L_2$, if L_2 is worse or equal in *all* criteria than L_1.)

The algorithm now maintains, like Dijkstra's algorithm, a priority queue Q of vertices. Keys for the priority queue entries must be chosen consistently among the criteria at every vertex u, e. g., one may choose the minimal arrival time of the labels in $B(u)$ [DW09b]. It is initialized by adding an initial label L_0 to $B(s)$ with all costs set to zero. In each iteration, the algorithm then extracts the vertex u with minimum key from Q and scans every arc $a = (u, v)$. However, when it scans a, it now creates a temporary bag $B_{\text{tent}}(v)$ by copying all labels from $B(u)$ and adding the cost of the arc a for every criterion to all labels in $B_{\text{tent}}(v)$. Then, $B_{\text{tent}}(v)$ is *merged* into $B(v)$: All labels from $B_{\text{tent}}(v)$ are copied into $B(v)$, thereby eliminating dominated labels on the fly. If *any* label from $B_{\text{tent}}(v)$ survived into $B(v)$, the vertex v is updated in the priority queue Q. The algorithm stops as soon as the priority queue runs empty.

If we are only interested in point-to-point queries to a target vertex t, we may make use of the following *target pruning*. Whenever the algorithm extracts a vertex u from Q, it checks if all labels in $B(u)$ are dominated by labels from $B(t)$. If this is the case, the algorithm prunes u, i. e., it does not scan outgoing arcs from u.

Note that this algorithm no longer scans vertices with increasing "distance", since they can no longer be totally ordered. Therefore, vertices may be inserted and extracted from Q multiple times, hence, the name label-*correcting* algorithm.

In the case of range queries on the time-dependent model, the same algorithm can be used. Here $B(u)$ corresponds to the connection points $P_f(u)$ of the travel time function representing optimal journeys from p_s to u. When an arc $a = (u, v)$ is relaxed, it takes the connection points $P_f(u)$ and computes tentative connection points by performing the link operation $f(u) \oplus f_a$ (cf. Section 3.1). Merging the tentative connection points into $P_f(v)$ exactly corresponds to the merge operation defined in Section 3.1.

For range queries, LC may (in addition to target pruning) employ the following stopping criterion: Whenever it scans a vertex u with associated connection points $P_f(u)$, it stops if the *lower bound* of the corresponding function $f(u)$ exceeds the *upper bound* of the travel time function $f(t)$ represented by the connection points $P_f(t)$ at the target vertex t. Note that, to ensure correctness, this requires $\underline{f}(u)$ as keys in the priority queue for the vertices u.

The running time of LC depends on the size of the bags $B(u)$ at each vertex. Merging two bags B_1 and B_2 requires time $\mathcal{O}(|B_1||B_2|)$, since it must check each pair $(L_1, L_2) \in B_1 \times B_2$ for domination. Unfortunately, the number of labels maintained during the algorithm's execution can be exponential in $|V|$ in theory [Han79], imposing a significant slowdown over Dijkstra's algorithm. However, for the optimization criteria we consider in this work, the algorithm remains practical.

Multi-Label-Correcting Algorithm. The Multi-Label-Correcting Algorithm (MLC), which has been considered in [PSWZ08, DMS08], works similar to LC in that it also maintains a *bag* of labels $B(u)$ with every vertex $u \in V$. However, instead of pushing entire bags when scanning an arc, it processes each label individually. More precisely, the algorithm keeps a priority queue Q of *labels* (instead of vertices). In addition, each label L in the priority queue Q also stores the vertex $u \in V$ to whom it belongs. The labels of Q are kept in arbitrary (but consistent) lexicographic order regarding the values of the associated criteria. Similarly to LC, the algorithm is initialized with empty bags for every vertex, except s, where it adds an initial label L_0 to $B(s)$. Moreover, it adds L_0 (together with s) to Q. In each iteration, it then extracts the label-vertex pair (L, u) with minimum (regarding the lexicographic order) key from Q and scans all arcs $a = (u, v) \in A$. For each, it creates a tentative label L_{tent} by adding the cost of the arc a to L and *merges* L_{tent} into $B(v)$, possibly dominating labels in $B(v)$. If L_{tent} is not dominated by any label from $B(v)$, the algorithm additionally adds it to Q (together with v). Note that every label that is removed from $B(v)$ (due to domination by L_{tent}) must also be removed from Q.

If we are only interested in point-to-point queries toward a vertex t, the target pruning rule of LC naturally carries over to MLC: Before inserting the tentative label L_{tent} into $B(v)$, the algorithm checks if L_{tent} is dominated by *any* label from the target bag $B(t)$. If this is the case, L_{tent} is simply discarded. See Figure 4.12 for an illustration of MLC in pseudocode.

In contrast to LC, handling range queries with MLC is conceptually easier and requires neither the link nor merge operations of travel time functions. Again, bags $B(u)$ correspond to the connection points $P_f(u)$ of the (partial) travel time function representing (tentative) journeys from s to u. MLC now works on the connection points individually (as described above) by using the following domination rule. Given two connection points q_1, q_2, the connection point q_1 dominates q_2, i. e., $q_1 \preccurlyeq q_2$ if and only if $\tau_{\text{dep}}(q_1) \geq \tau_{\text{dep}}(q_2)$ and $\tau_{\text{arr}}(q_1) \leq \tau_{\text{arr}}(q_2)$. Note that the stopping criterion from LC does not carry over to MLC, however, target pruning can still be applied.

In [DMS08] two additional improvements to MLC are proposed: The first, *hopping-reduction*, avoids propagating a label back to the vertex it originated from. More precisely, each label $L \in B(v)$ additionally keeps a *parent pointer* to the vertex u it originated from (i. e., the algorithm inserted L into $B(v)$ when it scanned an arc (u, v)). When the algorithm extracts L from Q at a later point in its execution, it may skip scanning the arc (v, u) (if it exists). The second improvement is *label forwarding*, which avoids using the priority queue for labels with no increase in cost: Whenever the algorithm scans an arc $a = (u, v)$ and creates a tentative label L_{tent} from L where $L = L_{\text{tent}}$, it does not insert L_{tent} (with v) into Q. (Note that L_{tent} would be extracted in the next iteration of the algorithm). Instead, it immediately proceeds with L_{tent}, scanning all arcs $(v, w) \in A$.

Similarly to LC, the MLC algorithm may also exhibit an exponential number of

```
// Input: Graph G = (V, A), source vertex s, target vertex t, departure time τ
// Side Effects: Pareto sets of labels B(u) at all vertices u ∈ V, if t = ⊥ or at t, otherwise
// Initialization of the algorithm
1  Q ← new PQueue()                                   // Create empty priority queue
2  B(·) ← ∅                                           // Create empty bags for every vertex
3  B(s) ← {L_0}                                       // Add initial label to bag at s
4  Q.Insert((L_0, s), Key(L_0))                       // Add initial label to priority queue
// Main loop
5  while not Q.Empty() do
6      (L, u) ← Q.ExtractMin()                        // Scan next label
7      forall the outgoing arcs a = (u, v) ∈ A do     // Scan outgoing arcs
8          L_tent ← L + Cost(a)                       // Create tentative label and add costs to it
9          forall the labels L' ∈ B(v) do             // Test for domination at v
10             if L' ≼ L_tent then break
11             if L_tent ≼ L' then
12                 B(v) ← B(v) \ {L'}
13                 Q.Delete((L', v))
14         forall the labels L' ∈ B(t) do             // Target pruning
15             if L' ≼ L_tent then break
16         if L_tent was not dominated then           // Merge tentative label into bag at v
17             B(v) ← B(v) ∪ {L_tent}
18             Q.Insert((L_tent, v), Key(L_tent))
```

Figure 4.12. Multi-Label-Correcting algorithm (MLC).

labels during execution, which yields the same (exponential in $|V|$) running time as LC in theory. However, for the criteria considered in this work, performance remains practical. An experimental comparison of LC and MLC on range queries (in a multimodal scenario) is conducted in [Bau12].

Layered Dijkstra. For a special case of the multicriteria problem, where one is interested in optimizing (besides arrival time) a second criterion that is discrete and only assumes a small number of different values, the following Layered Dijkstra (LD) algorithm may be more efficient [BJ04] than LC and MLC. We describe it using the number of transfers as exemplary criterion.

Therefore, let K be a bound on the number of transfers. During preprocessing, the graph is copied into K layers. Each transfer arc (in any layer) is rewired to point to the layer directly above. Then, running Dijkstra's algorithm from the source vertex s on the bottom layer results for each $k \leq K$ in an earliest arrival time that corresponds

to a journey having exactly k transfers for vertices on layer k. Instead of copying the graph, the algorithm uses an implicit representation of the layers. It, therefore, maintains an array of K labels at each vertex and reads/writes the k-th entry in layer k.

Moreover, to implement domination, a label at vertex u on layer k can be pruned if there exists a label with earlier arrival time at u on a layer lower than k. Similarly to implement target pruning for point-to-point queries, the label can be pruned if the target vertex has a label with smaller arrival time on any layer up to k. Note that we can drop the requirement for the bound K as input by dynamically extending the labels, whenever necessary.

Since LD essentially runs Dijkstra's algorithm on K copies of the graph G, the running time of this algorithm can be bounded by the number of layers K. Using a binary heap data structure as priority queue, thus, yields a running time of $\mathcal{O}(K(|V| + |A|) \log(K|V|))$.

4.5. Parallel Self-Pruning Connection Setting Algorithm

In this section we introduce our new parallel profile search algorithm for public transit networks. We start with a basic sequential algorithm for the general one-to-all setting in Section 4.5.1. Therefore, we first introduce the concept of *connection-setting* and show how some journeys dominate others. We then show in Section 4.5.2 the parallelization of our algorithm. In Section 4.5.3 we then present how it can also be utilized to accelerate point-to-point queries. A detailed review of our experiments is found in Section 4.5.4. We conclude with a summary in Section 4.5.5.

References. This section is based on [DKP09,DKP10,DKP12]. The publication [DKP10] was accepted at the 24th International Parallel and Distributed Processing Symposium (IPDPS'10) and [DKP12] appeared in the ACM Journal of Experimental Algorithmics, vol. 17, no. 1 in 2012. It is joint work with Daniel Delling and Bastian Katz.

Departing Connections. A crucial observation in public transit networks is the fact that each journey from a source stop p_s to any other stop has to begin with an elementary connection departing at p_s. Let this set of departing connections be denoted by $\mathcal{C}_{\text{dep}}(p_s)$ and defined as

$$\mathcal{C}_{\text{dep}}(p_s) := \{c \in \mathcal{C} \mid p_{\text{dep}}(c) = p_s\}. \tag{4.4}$$

A naïve and obvious way to compute the full travel time function $\text{dist}(p_s, \cdot, \cdot)$ would be to compute an earliest arrival query for each elementary connection $c \in \mathcal{C}_{\text{dep}}(p_s)$ with respect to its departure time $\tau_{\text{dep}}(c)$. However, such a connection does not necessarily contribute to the travel time function $\text{dist}(p_s, p_t, \cdot)$. A connection c_i with departure time $\tau_{\text{dep}}(c_i)$ may as well be *dominated* by a connection c_j with later

departure time $\tau_{\text{dep}}(c_j) > \tau_{\text{dep}}(c_i)$ in the following sense: If the earliest arrival time at p_t starting with c_j is not greater than the earliest arrival time starting with c_i, we can—and must, for the sake of correctness—*prune* the result of the search regarding connection c_i, since starting with c_i never yields the shortest travel time. Note that this observation implies that for any target stop $p_t \in \mathcal{S}$, the set of connection points $P(\text{dist}(p_s, p_t, \cdot))$ of the travel time function $\text{dist}(p_s, p_t, \cdot)$ is a subset of the set of connection points induced by $\mathcal{C}_{\text{dep}}(p_s)$ and their travel times to p_t. More precisely, the following holds:

$$\begin{aligned} P(\text{dist}(p_s, p_t, \cdot)) \subseteq \{(\tau, \ell) \mid \text{there is } c \in \mathcal{C}_{\text{dep}}(p_s) \text{ such that} \\ \tau = \tau_{\text{dep}}(c) \text{ and} \\ \ell = \text{dist}(p_s, p_t, \tau_{\text{dep}}(c))\}. \end{aligned} \tag{4.5}$$

The problem to run $|\mathcal{C}_{\text{dep}}(p_s)|$ earliest arrival queries and then pruning dominated connections from $\text{dist}(p_s, p_t, \cdot)$ afterwards is an embarrassingly parallel problem. Going much further, we show how to extend the above observation to obtain a pruning rule that we call *self-pruning*. It can be applied to eliminate "unnecessary" connections as soon as possible. Thereby, we use self-pruning within the restricted domain of each single thread, but also take advantage of communication between the different threads yielding a rule we call *inter-thread-pruning*. Therefore, we require a fixed assignment of the departing connections to the processors where each processor handles a set of connections simultaneously.

The outline of our new parallel algorithm is as follows: First, we partition the set $\mathcal{C}_{\text{dep}}(p_s)$ of departing connections to a given set of processors. Second, every processor runs a single thread applying our main sequential profile search algorithm restricted to its subset of departing connections. In a third step, the partial results by the different threads are combined, thereby eliminating dominated connections that could not be pruned earlier, a step we will refer to as *connection reduction*.

4.5.1. The Main (Sequential) Algorithm

From the point of view of a single processor that has some subset of $\mathcal{C}_{\text{dep}}(p_s)$ as input, it basically makes no difference to the profile search algorithm that some of the connections are ignored. We simply obtain $\text{dist}_k(p_s, \cdot, \cdot)$ restricted to the connections assigned to the particular processor k. Hence, we describe the main algorithm as if it was a purely sequential profile search algorithm and turn towards the parallel issues, like merging the results from each processor, the way we partition the departing connections $\mathcal{C}_{\text{dep}}(p_s)$ and our inter-thread-pruning rule, afterwards.

The naive approach of running a separate earliest arrival query for each $c \in \mathcal{C}_{\text{dep}}(p_s)$ by Dijkstra's algorithm (cf. Section 4.4.1) would require an empty priority queue for every connection c. By contrast, our algorithm maintains a *single* priority queue and handles all of its connections simultaneously. Moreover, we use tentative arrival times

as keys (instead of distances). By these means, we enable both the connection-setting property as well as our self-pruning rule.

Initialization. At first, the set $\mathcal{C}_{\text{dep}}(p_s)$ of departing conncetions is determined and ordered non-decreasingly by the departure times of the elementary connections in $\mathcal{C}_{\text{dep}}(p_s)$. Thus, we may say that a connection $c_i \in \mathcal{C}_{\text{dep}} p_s$ has *index* i according to the ordering of $\mathcal{C}_{\text{dep}}(p_s)$. We may use the term index and departing connection interchangeably in the following. The elements of the priority queue are pairs (u, i) where the first entry depicts a vertex $u \in V$ and the second entry a connection index $0 \leq i < |\mathcal{C}_{\text{dep}}(p_s)|$. For each vertex $u \in V$ and for each connection i a label $L(u, i)$ is assigned which depicts the (tentative) *arrival time* at u when for a journey that starts with connection i. In the beginning, all label $L(u, i)$ are initialized with ∞. Then, for each connection $c_i \in \mathcal{C}_{\text{dep}}(p_s)$ we insert (u_r, i) with key $\tau_{\text{dep}}(c_i)$ into the priority queue, where u_r depicts the corresponding route vertex of c_i at stop p_s in the graph G. Note that in the beginning the "arrival time" $L(u_r, i)$ equals the departure time $\tau_{\text{dep}}(c_i)$.

Connection-Setting. Like Dijkstra's algorithm, we subsequently extract the queue element (u, i) with minimum key and assign key(u, i) as the final arrival time to $L(u, i)$. Then, for each arc $a = (u, v) \in A$, we compute a tentative label $L'(v, i)$ at vertex v by evaluating the arc a at time $L(u, i)$, i. e., we set $L'(v, i) := L(u, i) + f_a(L(u, i))$ (for connection i). If v has not yet been discovered using connection i, we insert (v, i) into the priority queue with key$(v, i) := L'(v, i)$, otherwise, the element (v, i) is already in the queue and we set key(v, i) to min(key$(v, i), L'(v, i)$). Note that the following holds for every connection i: When a queue item (u, i) is scanned, the label $L(u, i)$ is final, thus, the label-setting property holds with respect to each connection i. We call this property *connection-setting property*. The algorithm stops as soon as the priority queue runs empty. We end up with labels $L(u, i)$ for each vertex $u \in V$ and each connection $0 \leq i < |\mathcal{C}_{\text{dep}}(p_s)|$. Each label depicts the arrival time at u when starting with the i-th connection from p_s.

We stress out two things. First, although the computation is done for all connections simultaneously, they can be regarded as independent, since the labels and the queue items refer to a specific connection throughout the algorithm. Second, the original variant of Dijkstra's algorithm uses distances instead of arrival times as keys. However, this has no impact on the correctness of the algorithm: For each connection the distance can be obtained by subtracting the respective departure time from the arrival time, which is constant for all vertices.

Connection Reduction and Self-Pruning. For each vertex $u \in V$ the final labels $L(u, \cdot)$ induce a set of connection points P by

$$P := \{(\tau_{\text{dep}}(c_i), \delta(\tau_{\text{dep}}(c_i), L(u, i))) \mid c_i \in \mathcal{C}_{\text{dep}}(p_s)\}. \tag{4.6}$$

Unfortunately, the travel time function f represented by P does not account for domination of connections and hence does not necessarily fulfill the FIFO property (cf. Section 4.3.3). Formally, for two connection points $(\tau_i, \ell_i), (\tau_j, \ell_j) \in P$ with $j > i$ it is possible that $\tau_j + \ell_j \leq \tau_i + \ell_i$. The aforementioned *connection reduction*, which remedies this issue *at the end of the algorithm*, reduces P to obtain $P(\text{dist}(S, T, \cdot))$ by removing those connection points which are dominated by another connection point with a *later* departure time and an *earlier* arrival time.

More precisely, the algorithm scans P backwards and keeps track of the minimum arrival time τ^*_{arr} along the way induced by the connection with index i^*, i. e., $\tau^*_{\text{arr}} := \tau_{\text{dep}}(i^*) + \tau_{\text{tra}}(i^*)$. Each time it scans a connection point $j < i^*$ with an arrival time $\tau_{\text{arr}}(j) \geq \tau^*_{\text{arr}}$, the connection point is deleted. The remaining connection points are exactly those of $P(\text{dist}(p_s, p_t, \cdot))$.

Performing this connection reduction after termination of the algorithm results in the computation of many unnecessary connections and, therefore, many unnecessary queue operations. Recall that the keys in the priority queue are arrival times. Thus, we propose a more sophisticated approach to eliminate dominated connections *during* the algorithm: We introduce a *vertex label* maxconn: $V \to \{0, \ldots, |\mathcal{C}_{\text{dep}}(p_s)| - 1\}$ depicting for a vertex $u \in V$ the highest connection index with which the vertex u has been scanned so far. Now, each time the algorithm extracts a queue element (u, i) with $L(u, i) := \text{key}(u, i)$, it checks if $i > \text{maxconn}(u)$ holds. If this is *not* the case, the vertex u has already been scanned earlier but with a later connection (remember that $j > i \Rightarrow \tau_{\text{dep}}(c_j) \geq \tau_{\text{dep}}(c_i)$), thus, implying $L(u, j) \leq L(u, i)$. Therefore, the current connection does not contribute to the solution, and the algorithm prunes the connection i at u, i. e., it does not relax outgoing arcs from u. Moreover, it sets $L(u, i) := \infty$, depicting that no journey beginning with the i-th connection reaches u. In case that $i > \text{maxconn}(u)$, the algorithm updates maxconn(u) to i and continues with scanning the outgoing arcs of u, regularly.

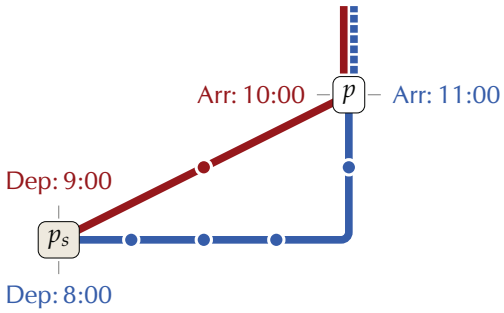


Figure 4.13. Illustrating domination.

Obviously, by applying self-pruning, the set of connection points $P(\text{dist}(p_s, u, \cdot))$ at each vertex u induced by $L(u, \cdot)$ fulfills the FIFO property automatically (labels with $L(u, i) = \infty$ have to be ignored).

Figure 4.13 illustrates domination between connections. The red route is an express route, the blue one a local route. At (a vertices belonging to) stop p, the blue connection is pruned by the red connection, since it has an earlier arrival time and a later departure time at p_s. Also for stops beyond p, only the red connection is optimal.

Theorem 1. *Applying self-pruning is correct.*

Proof. Let $u \in V$ be an arbitrary vertex. We show that no optimal connection to u has been pruned by contradiction. Let $L(u,i)$ be the arrival time at u of the (optimal) i-th connection and assume that i has been pruned at u. Let j denote the connection which was responsible for pruning i. Then, it holds that $L(u,j) \leq L(u,i)$. Moreover, since j pruned i, it holds that $j > i$, which implies $\tau_{\text{dep}}(c_j) \geq \tau_{\text{dep}}(c_i)$. Therefore, it holds that $\delta(\tau_{\text{dep}}(c_j), L(u,j)) \leq \delta(\tau_{\text{dep}}(c_i), L(v,i))$. This is a contradiction to i being optimal: Using the j-th connection results in an earlier arrival time at u by departing later at p_s. ∎

Putting things together, pseudocode of the complete (sequential) algorithm can be found in Figure 4.14.

4.5.2. Parallelization

Unlike the trivial parallelization that would assign a connection $c \in \mathcal{C}_{\text{dep}}(p_s)$ for an arbitrary idle processor which then runs Dijkstra's algorithm on c, our algorithm needs a fixed assignment of the connections to the processors beforehand. Let P denote the number of processors available. In a first step, we partition $\mathcal{C}_{\text{dep}}(p_s)$ into P subsets where each thread k runs our main algorithm on its restricted subset $\mathcal{C}^k_{\text{dep}}(p_s)$.

After each thread terminates, we obtain partial travel time functions $\text{dist}^k(p_s, \cdot, \cdot)$ restricted to the connections that were assigned to thread k. Thus, the master thread merges the labels $L^k(u, \cdot)$ of each thread k to a common label $L(u, \cdot)$, thereby, preserving the ordering of the connections. This can be done by a linear sweep over the labels. Note that the common label $L(u, \cdot)$ does not necessarily fulfill the FIFO property, since we do not self-prune between threads (so far). For that reason, the connection points $P(p_s, p_t, \cdot))$ of the final distance function are obtained by reducing the connection points induced by the common label $L(u, \cdot)$ with our connection reduction method described above. The pseudocode of the main parallel algorithm is presented in Figure 4.15.

Choice of the Partition. The speedup achieved by the parallelization of our algorithm depends on the partitioning of $\mathcal{C}_{\text{dep}}(p_s)$. As the overall computation time is dominated by the thread with the longest computation time (for computing the final travel time function, all threads have to be in a finished state), nearly optimal parallelism would be achieved if all threads share the same amount of queue operations, thus, approximately sharing the same computation time. However, this figure is not known beforehand, which requires us to partition $\mathcal{C}_{\text{dep}}(p_s)$ heuristically. We propose the following simple methods.

The *equal time-slots* method partitions the complete time interval Π into P intervals of equal size. While this can be computed easily, the sizes of $\mathcal{C}_{\text{dep}}(p_s)^i$ turn out

```
// Input: Graph G = (V, A), source stop p_s, outgoing connections C_dep(p_s)
// Side Effects: Distance labels L(·,·) for each vertex and connection
1  Q ← new PQueue()                                          // Create empty priority queue
2  maxconn(·) ← −∞                                           // Initialization
3  L(·,·) ← ∞
4  discovered(·,·) ← false
5  Sort(C_dep(p_s))                                          // Order outgoing connections by departure time
6  forall the connections c_i ∈ C_dep(p_s) do                // Add route vertices at p_s to queue
7      r ← route vertex belonging to c_i
8      Q.Insert((r, i), τ_dep(c_i))
9      discovered(r, i) ← true

   // Main loop
10 while not Q.Empty() do
11     (u, i) ← Q.ExtractMin()                               // Scan next vertex/connection
12     if maxconn(u) > i then                                // Self-pruning rule
13         L(u, i) ← ∞
14         continue
15     else
16         maxconn(u) ← i

17     forall the outgoing arcs a = (u, v) ∈ A do            // Scan outgoing arcs
18         L'(v, i) ← L(u, i) + f_a(L(u, i))                 // Create tentative label
19         if not discovered(v, i) then
20             Q.Insert((v, i), L'(v, i))                    // Insert tuple into priority queue
21             L'(v, i) ← L(v, i)
22             discovered(v, i) ← true
23         else if L'(v, i) < Q.Key((v, i)) then
24             Q.DecreaseKey((v, i), L'(w, i))               // Update key in priority queue
25             L'(v, i) ← L(v, i)
```

Figure 4.14. Pseudocode of the Self-Pruning Connection-Setting Algorithm (SPCS).

```
// Input: Graph G = (V, E), source stop p_s, outgoing connections C_dep(p_s), P processors
// Side Effects: Distance labels L(·,·) for each vertex and connection
1  {C^1_dep(p_s), ..., C^P_dep(p_s)} ← Partition(C_dep(S))                  // Initialization
2  for k ← 1...P do in parallel                                             // Parallel computation
3      L^k(·,·) ← ∞
4      SPCS(C^k_dep(p_s))                                                   // Invoke the sequential SPCS algorithm
5  L(·,·) ← Merge(L^1(·,·), ..., L^P(·,·))                                  // Merge labels from threads
   // Connection reduction
6  forall the u ∈ V do
7      last ← ∞
8      for i ← |C_dep(p_s)| ... 1 do
9          if L(u,i) < last then
10             last ← L(u,i)
11         else
12             L(u,i) ← ∞
```

Figure 4.15. Parallel Self-Pruning Connection-Setting algorithm (PSPCS).

to be very unbalanced, at least in our scenario. The reason for this is that connections in $\mathcal{C}_{\text{dep}}(p_s)$ are not distributed uniformly over the day due to rush hours and operational breaks at night.

The *equal number of connections* method tries to improve on that by partitioning the set $\mathcal{C}_{\text{dep}}(p_s)$ into P sets of equal size (i. e., containing equally many subsequent elementary connections). This is also very easy to compute and improves over the equal time-slots method regarding the balance. Besides these simple heuristics, in principle, more sophisticated clustering methods like k-Means [Mac67] can be applied. However, our experimental evaluation (cf. Section 4.5.4) shows that the improvement in query performance is negligible compared to the simple methods, thus, we use the equal number of connections method as a reasonable compromise. We stress that for the correctness of our algorithm it is not necessary to partition $\mathcal{C}_{\text{dep}}(p_s)$ into cells of subsequent connections. However, it is intuitive to see that the self-pruning rule is most effective on neighboring (regarding the departure time) connections.

Pruning Between Threads. When computing partial travel time functions *independently* in parallel, the speedup gained by self-pruning may decrease, since a connection j cannot prune a connections i, if i is assigned to a different thread than j. Thus, with an increasing number of threads, the effect achieved of self-pruning vanishes to the extreme point where the number of threads equals the number of connections in $\mathcal{C}_{\text{dep}}(p_s)$. In this case, our algorithm basically corresponds to computing $|\mathcal{C}_{\text{dep}}(p_s)|$

```
// Input: Thread number k, number of processors P, ...
1  ...
2  minarr^k(·) ← ∞
3  ...
4  while not Q.Empty() do
5      ...
       // Inter-thread-pruning rule
6      if there is l with k < l ≤ P for which minarr^l(u) ≤ L(u,i) then
7          L(u,i) ← ∞
8          continue
9      minarr^k(u) ← min(minarr^k(u), L(u,i))
10     ...
```

Figure 4.16. Inter-thread-pruning rule for PSPCS. To make the figure less cluttered, only the relevant parts of the total algorithm are shown.

earliest arrival queries in parallel—without any pruning.

To remedy this issue, the self-pruning rule can be augmented in order to make use of dominating connections across different threads. In the case that the partitioning of $\mathcal{C}_{\text{dep}}(p_s)$ is chosen such that each cell $\mathcal{C}_{\text{dep}}(p_s)^k$ only contains subsequent connections, we may define a total ordering on the cells by $\mathcal{C}_{\text{dep}}(p_s)^k \prec \mathcal{C}_{\text{dep}}(p_s)^l$ if for all connections $c \in \mathcal{C}_{\text{dep}}(p_s)^k$ and all connections $c' \in \mathcal{C}_{\text{dep}}(p_s)^l$ it holds that $\tau_{\text{dep}}(c) \leq \tau_{\text{dep}}(c')$. Without loss of generality, let $k < l \Leftrightarrow \mathcal{C}_{\text{dep}}(p_s)^k \prec \mathcal{C}_{\text{dep}}(p_s)^l$. We introduce an additional vertex label $\text{minarr}^k \colon V \to \Pi$ for each thread k that depicts for every vertex u the earliest arrival time at u using connections assigned to the k-th thread. In the beginning, the algorithm initializes $\text{minarr}^k(u)$ to infinity and updates $\text{minarr}^k(u) := \min(\text{minarr}_k(u), \tau_{\text{arr}}(u,i))$ each time thread k scans the vertex u for some connection i. Then, in addition to the self-pruning rule, we propose the following *inter-thread-pruning* rule: Each time the algorithm scans a queue element (u,i) with $\tau_{\text{arr}}(u,i) = \text{key}(u,i)$ in thread k, it checks if there exists a thread l with $l > k$ for which $\text{minarr}^l(u) \leq \tau_{\text{arr}}(u,i)$. If this is the case, it holds by the total ordering of the partition cells that there exists a connection j assigned to thread l with $\tau_{\text{dep}}(c_j) \geq \tau_{\text{dep}}(c_i)$ but $\tau_{\text{arr}}(u,j) \leq \tau_{\text{arr}}(u,i)$. In other words, connection i assigned to thread k is dominated by a connection j assigned to thread l. Thus, the algorithm prunes i at u the same way it does for self-pruning, i. e., it does not relax outgoing arcs from u for connection i. Correctness of this rule can be proven analogue to the the self-pruning rule described earlier.

In a shared memory setup like in multicore servers, the values of $\text{minarr}^k(\cdot)$ can be communicated through the main memory, thus, not imposing a significant overhead to the algorithm. Moreover, for practical use it is sufficient to only check a constant

number x of threads $\{k+1, \ldots, k+x\}$, since dominating connections are less likely to be "far in the future", i. e., assigned to threads $l \gg k$. Furthermore, we like to mention that the inter-thread-pruning rule does not *guarantee* pruning of dominated connections since the priority queue is not shared across threads. However, in most cases connections j with small arrival times prune connections i with high arrival time with respect to their particular thread. Hence, j is likely to be scanned before i in the parallel execution, thus, enabling pruning of i. An pseudocode illustration of the inter-thread-pruning rule is presented in Figure 4.16.

4.5.3. Point-to-Point Queries

Dijkstra's algorithm can be accelerated by precomputing auxiliary data as soon as we are only interested in point-to-point queries [DSSW09a]. In this section, we present how some of the ideas, for example, the *stopping criterion*, map to our new algorithm. Moreover, we show how the precomputation of certain journeys improves the performance of our algorithm. The enhancements introduced in this section refer to the sequential algorithm (cf. Section 4.5.1). Thus, all results translate to our parallel algorithm naturally. Also note that they require a target stop as input, in particular, they do not accelerate one-to-all queries.

Stopping Criterion

For point-to-point queries, Dijkstra's algorithm can stop the query as soon as the target node has been extracted from the priority queue. In our case, i. e., stop-to-stop, we can abort the query as soon as the target stop p_t has its final label $L(p_t, i)$ for all i assigned. This is achieved as follows. The algorithm maintains an index T_m, initialized with $-\infty$. Whenever it scans a connection i at the target stop p_t, it sets $T_m := \max\{i, T_m\}$. Then, the algorithm may prune all queue entries $(u, i) \in Q$ for which $i \leq T_m$ holds (at *any* vertex u). The query terminates as soon as the queue is empty.

Theorem 2. *The stopping criterion is correct.*

Proof. We need to show that no queue entry $q = (u, i) \in Q$ with $i \leq T_m$ can improve the arrival time at p_t for the connection i. Let, therefore, $q' = (u', i')$ be the responsible entry that has set T_m. Since $i \leq T_m$ holds, we know that regarding the departure times of the connections, $\tau_{\text{dep}}(c'_i) \geq \tau_{\text{dep}}(c_i)$ must hold. Moreover, since q is scanned after q', we know that $L(u', i') \leq L(u, i)$ must hold. In other words, it does not pay off to continue journey i at stop p. ∎

Pruning with a Distance Table

Next, we show how to accelerate out point-to-point algorithm by pruning with the help of a distance table. Since a distance table computed directly on the model

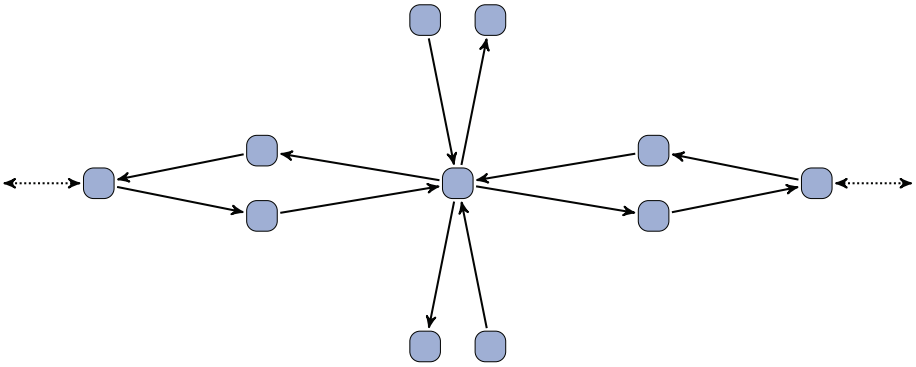


Figure 4.17. The super station graph $\tilde{G}$ corresponding to the network depicted by Figure 4.10.

graph G would be too large to be practical, we use the smaller *super stop network* to compute the distance table. Intuitively, super stops are obtained by merging stops that are connected by footpaths.

Constructing Super Stops. Consider the *foot graph* $G_{\text{foot}} = (\mathcal{S}, \mathcal{F})$ whose vertices are exactly the stops of the timetable and arcs correspond to footpaths. As mentioned in Section 4.3, G_{foot} is composed of small connected components of stops near the same intersection of the underlying road network. Thus, we use G_{foot} to obtain a *super stop graph* $\tilde{G} = (\tilde{\mathcal{S}}, \tilde{A})$ in the following way. For each connected component in G_{foot} we create a *super stop* $\tilde{p}$ in $\tilde{\mathcal{S}}$. An arc $(\tilde{p}_i, \tilde{p}_j)$ is contained in $\tilde{A}$ if and only if there exists an elementary connection from any of the stops inside $\tilde{p}_i$ to any of the stops inside $\tilde{p}_j$. We use $\tilde{\mathcal{S}}(p)$ to refer to the super stop of a stop $p \in \mathcal{S}$. See Figure 4.17 for the super stop graph obtained from the network depicted in Figure 4.10.

Furthermore, for our pruning rule, we require the notion of the *diameter* of a super stop $\tilde{p}$. It is defined as the length of the longest shortest path inside a component of G_{foot}, but additionally takes the minimum change times at its respective source and target stops into account. Formally, let $\text{dist}(p_i, p_j)$ denote the shortest path distance between two stops p_i and p_j in G_{foot}. Then we define

$$\text{diam}(\tilde{p}) := \max_{p_i, p_j \in \tilde{p}} \{\tau_{\text{ch}}(p_i) + \text{dist}(p_i, p_j) + \tau_{\text{ch}}(p_j)\}. \tag{4.7}$$

Think of the diameter as an upper bound on the time one can spend walking inside a super stop.

Hub Stops. We are now given a subset $\tilde{\mathcal{S}}_{\text{hub}} \subseteq \tilde{\mathcal{S}}$ of super stops, called *hub super stops* (think of them as important hubs in the network) and a distance table $\mathcal{D} : \tilde{\mathcal{S}}_{\text{hub}} \times \tilde{\mathcal{S}}_{\text{hub}} \times \Pi \to \mathbb{Z}_{\geq 0}$. The distance table returns, for each pair of super stops $\tilde{p}_i, \tilde{p}_j \in \tilde{\mathcal{S}}_{\text{hub}}$, the quickest way of getting from $\tilde{p}_i$ to $\tilde{p}_j$ at time $\tau \in \Pi$, i. e., the earliest possible arrival time at $\tilde{p}_j$ for any of the combinations of a stop inside $\tilde{p}_i$ and a stop inside $\tilde{p}_j$.

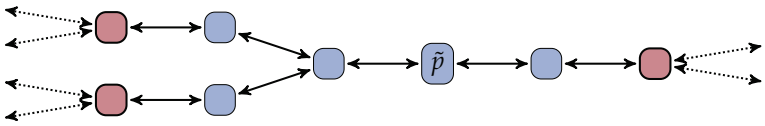


Figure 4.18. Local and via super stops of a super stop $\tilde{p}$. Local super stops are drawn blue, while via super stops are marked bold and red.

Note that we do not consider the diameters of $\tilde{p}_i$ and $\tilde{p}_j$ here. In other words, the distance table returns a *lower bound* on the distance between $\tilde{p}_i$ and $\tilde{p}_j$ at time τ.

Before explaining the pruning rule in detail, we need the additional notion of *local* and *via super stops*. The set of local super stops $\text{local}(\tilde{p}) \subseteq \tilde{\mathcal{S}}$ of an arbitrary super stop $\tilde{p} \in \tilde{\mathcal{S}}$ includes all super stops $\tilde{p}'$ such that there is a simple path from $\tilde{p}'$ to $\tilde{p}$ that contains only non-hub super stops in the super stop graph $\tilde{G}$. The set of hub super stops that are adjacent to at least one local super stop of $\tilde{p}$ are called the *via super stops* of $\tilde{p}$, denoted by $\text{via}(\tilde{p}) \subseteq \tilde{\mathcal{S}}_{\text{hub}}$. They basically separate $\tilde{p} \cup \text{local}(\tilde{p})$ from any other super stop in $\tilde{G}$. Figure 4.18 gives a small example. In the special case of $\tilde{\mathcal{S}}$ being a hub super stop itself, we set $\text{local}(\tilde{p}) = \emptyset$ and $\text{via}(\tilde{\mathcal{S}}) = \{\tilde{p}\}$.

Applying the Distance Table. In the following, we call a p_s–p_t (with respective super stops $\tilde{p}_s$ and $\tilde{p}_t$) query *local*, if $\tilde{p}_s \in \text{local}(\tilde{p}_t)$; otherwise the query is called *global*. Note that an optimal journey of a global query must contain a via super stop of $\tilde{p}_t$. We accelerate global p_s–p_t queries by maintaining an upper bound $\mu_{i,j}$ (initialized with ∞) for each connection i and each via super stop $\tilde{p}_j$ from $\text{via}(\tilde{p}_t)$. Whenever the algorithm extracts a queue entry $q = (u, i)$ with $\tilde{p}(v) \in \tilde{\mathcal{S}}_{\text{hub}}$, it sets

$$\mu_{i,j} := \min\{\mu_{i,j}, \mathcal{D}(\tilde{p}(u), \tilde{p}_j, L(u,i) + \text{diam}(\tilde{p}(u))) + \text{diam}(\tilde{p}_j)\} \tag{4.8}$$

for all $\tilde{p}_j \in \text{via}(\tilde{p}_t)$. In other words, $\mu_{i,j}$ depicts an upper bound on the earliest trip one can get at $\tilde{p}_j$, even if it involved a transfer (and potential walk) at $\tilde{p}_j$. So, the algorithm prunes the search for q if

$$\text{for all } \tilde{p}_j \in \text{via}(\tilde{p}_t)\colon \mathcal{D}(\tilde{p}(u), \tilde{p}_j, L(u,i)) > \mu_{i,j} \tag{4.9}$$

holds. In other words, the search is pruned at u for a connection i if the path through $\tilde{p}(u)$ is provably not important for the optimal journey to *any* via stop of $\tilde{p}_j \in \text{via}(\tilde{p}_t)$. Figure 4.19 gives a small example.

Theorem 3. *Pruning based on a distance table is correct.*

The following proof is split into several lemmas and follows the intuition that arriving at a time earlier than $\mu_{i,j}$ at $\tilde{p}_j$ ensures getting the optimal trip towards p_t. Moreover, when the algorithm prunes at u, the path through u yields a later arrival time at $\tilde{p}_j$ than $\mu_{i,j}$. Thus, the path at u can be pruned, since it is no improvement over the path corresponding to $\mu_{i,j}$.

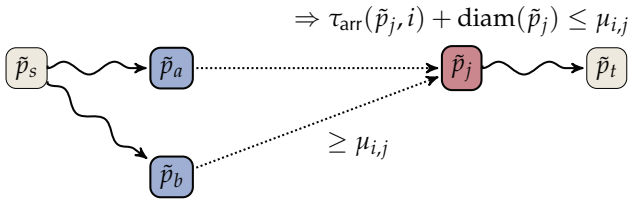


Figure 4.19. Example for pruning via a distance table, given an p_s–p_t query. The super stops $\tilde{p}_a$ and $\tilde{p}_b$ are hub super stops, and $\tilde{p}_j$ are the via super stop of $\tilde{p}_t$. When scanning a vertex of super stop $\tilde{p}_a$, we obtain that the arrival time at $\tilde{p}_j$ plus the diameter at $\tilde{p}_j$ is smaller or equal to $\mu_{i,j}$. Hence, the algorithm prunes the query at $\tilde{p}_b$ if the lower bound obtained from the distance table yields an arrival time at $\tilde{p}_j$ greater than $\mu_{i,j}$.

We prove the overall correctness by showing the correctness for each connection i separately. Thus, let i be a fixed connection index and $P = [p_s, \ldots, p_t]$ the shortest path of a global p_s–p_t query of connection i. Note that if p_s-p_t is a local query, no pruning is applied and, hence, there is nothing to prove.

Now, let $\tau^*_{\text{arr}}(p_t, i)$ denote the optimal arrival time at p_t of the path P (i. e., by starting with connection i). Moreover, let $\tilde{p}_t$ be the corresponding super stop of the target stop p_t. To show the main theorem, we prove a series of lemmas first.

Lemma 2. *For all tuples $(u, \tilde{p}_j) \in V \times \text{via}(\tilde{p}_t)$ with $\tilde{p}(u) \in \tilde{\mathcal{S}}_{hub}$ it holds that*

$$\tau^*_{arr}(p_t, i) \leq \underbrace{\mathcal{D}\big(\tilde{p}(u), \tilde{p}_j, L(u,i) + \text{diam}(\tilde{p}(u))\big) + \text{diam}(\tilde{p}_j)}_{=:\,\mu_{i,u,j}} + \text{dist}(\tilde{p}_j, p_t, \mu_{i,u,j}). \tag{4.10}$$

Proof. Assume that the equation is false and the right hand side yields an arrival time at p_t which is earlier than $\tau^*_{\text{arr}}(p_t, i)$. Then, the path induced by the right hand side of the equation yields a shorter path to p_t, which is a contradiction to $\tau^*_{\text{arr}}(p_t, i)$ being optimal. ∎

This proves that using the distance table via $\tilde{p}_j$ at any vertex u yields an upper bound on the arrival time at p_t (for connection i). Since this is true at all vertices $u \in V$ (for which $\tilde{p}(u) \in \tilde{\mathcal{S}}_{\text{hub}}$), the following corollary follows immediately.

Corollary 1. *Let*

$$\mu_{i,j} := \min_{\substack{u \in V \\ \tilde{p}(u) \in \tilde{\mathcal{S}}_{hub}}} (\mu_{i,u,j}). \tag{4.11}$$

Then it holds that $\tau^*_{arr}(p_t, i) \leq \mu_{i,j} + \text{dist}(\tilde{p}_j, T, \mu_{i,j})$.

Note that in the algorithm $\mu_{i,j}$ is maintained exactly the way it is defined in Lemma 2, and the minimum operation is applied iteratively each time it scans a

vertex u for which $\tilde{p}(v) \in \tilde{\mathcal{S}}_{\text{hub}}$ holds. Hence, the inequality of Corollary 1 holds in the algorithm, as well.

Next, consider the combined shortest p_s–u–$\tilde{p}_j$–p_t path of connection i and arrival time $\tau_{\text{arr}}^{(j)}(p_t, i)$ at p_t. We lower-bound $\tau_{\text{arr}}^{(j)}(p_t, i)$ by the distance table in the following lemma.

Lemma 3. *For all tuples $(u, \tilde{p}_j) \in V \times \text{via}(\tilde{p}_t)$ with $\tilde{p}(u) \in \tilde{\mathcal{S}}_{hub}$ it holds that*

$$\tau_{arr}^{(j)}(p_t, i) \geq \underbrace{\mathcal{D}(\tilde{p}(u), \tilde{p}_j, L(u, i))}_{=: \gamma_{i,u,j}} + \text{dist}(\tilde{p}_j, p_t, \gamma_{i,u,j}) \tag{4.12}$$

where $\tau_{arr}^{(j)}(p_t, i)$ depicts the arrival time of the combined shortest p_s–u–$\tilde{p}_j$–p_t path.

Proof. Assume that the right hand side of the equation evaluates to $\tau_{\text{arr}}^{(j)'}(p_t, i)$ with $\tau_{\text{arr}}^{(j)'}(p_t, i) > \tau_{\text{arr}}^{(j)}(p_t, i)$. Then, because both $\mathcal{D}(\tilde{p}(u), \tilde{p}_j, \cdot)$ and $\text{dist}(\tilde{p}_j, p_t, \cdot)$ are fulfilling the FIFO property, the departure time τ of $\mathcal{D}(\tilde{p}(u), \tilde{p}_j, \tau)$ of the path corresponding to $\tau_{\text{arr}}^{(j)'}(p_t, i)$ on the left hand side of the inequation has to be strictly smaller than $\tau_{\text{arr}}(u, i)$ at u. But, this cannot be true, since the path induced by $\tau_{\text{arr}}^{(j)}(p_t, i)$ is assumed to be the shortest path. ■

Intuitively, Lemma 3 proves that any valid (shortest) p_s–p_t path that goes via u and $\tilde{p}_j$ has to be at least as long as the "path" that ignores walking times at both $\tilde{p}(u)$ and $\tilde{p}_j$ (and basically acts as if one could catch any trip at $\tilde{p}(u)$ and $\tilde{p}_j$ instantaneously).

Next, we establish that, when we apply our pruning rule during the algorithm, we do not prune a path that is important (i. e., we only prune paths which are provably not shortest to p_t).

Lemma 4. *Let $u \in V$ be a vertex with $\tilde{p}(u) \in \tilde{\mathcal{S}}_{hub}$, and let $\gamma_{i,u,j} > \mu_{i,j}$. Then*

$$\gamma_{i,u,j} + \text{dist}(\tilde{p}_j, p_t, \gamma_{i,u,j}) \geq \mu_{i,j} + \text{dist}(\tilde{p}_j, p_t, \mu i, j) \tag{4.13}$$

holds.

Proof. This follows immediately from the FIFO property of $\text{dist}(\tilde{p}_j, p_t, \cdot)$. ■

We now conclude our proof of the main Theorem 3. Hence, let $u \in V$ be a vertex with $\tilde{p}(u) \in \tilde{\mathcal{S}}_{\text{hub}}$, where the pruning rule is potentially applied by the algorithm. Then from Lemmas 3, 4, and Corollary 1 we obtain for a via super stop $\tilde{p}_j \in \text{via}(\tilde{p}_t)$ that

$$\gamma_{i,u,j} > \mu_{i,j} \Rightarrow \tau_{\text{arr}}^{(j)}(p_t, i) \geq \underbrace{\mu_{i,j} + \text{dist}(\tilde{p}_j, p_t, \mu_{i,j})}_{=: \psi} \geq \tau_{\text{arr}}^{*}(p_t, i). \tag{4.14}$$

Since our algorithm keeps track of $\mu_{i,j}$ as the minimum over all $\mu_{i,\tilde{p},j}$ with $\tilde{p} \in \tilde{\mathcal{S}}_{\text{hub}}$, the path which corresponds to $\mu_{i,j}$ is not pruned. Hence, at the point where u is

pruned, a path with arrival time ψ toward $\tilde{p}_j$ is guaranteed to be found. Since u is only pruned if Equation 4.13 holds for *all* $\tilde{p}_j \in \text{via}(\tilde{p}_t)$, it follows that u is not on the path P, thus, u is not important for the shortest p_s–p_t path. ∎

Computing Via Super Stops. The query algorithm determines the via super stops of $\tilde{p}_t$ on-the-fly: During the initialization phase it runs a depth-first search on the *reverse* super stop graph from $\tilde{p}_t$, pruning the search at stops $\tilde{p} \in \tilde{\mathcal{S}}_{\text{hub}}$. Any super stop $\tilde{p} \in \tilde{\mathcal{S}}_{\text{hub}}$ touched during the depth-first search is added to $\text{via}(\tilde{p}_t)$. Note that the algorithm distinguishes local from global queries when computing $\text{via}(\tilde{p}_t)$: As soon as the depth-first search visits $\tilde{p}_s$, the query is local, otherwise it is global.

Selecting Hub Super Stops

The efficiency of pruning via a distance table highly depends on which super stops are selected for $\tilde{\mathcal{S}}_{\text{hub}}$. In [SWW00], the authors propose to identify important stops by a given "importance" value provided by the input. However, such values are not available for all inputs. Hence, we compute importance values heuristically. Consider the aforementioned super stop graph $\tilde{G}$. We augment $\tilde{G}$ with constant arc weights $\ell(\tilde{p}_i, \tilde{p}_j)$. Therefore, consider all connection poins (or, equivalently, elementary connections) that go from any stop inside $\tilde{p}_i$ to any stop inside $\tilde{p}_j$, denoted by $\mathcal{C}(\tilde{p}_i, \tilde{p}_j)$. Then, we define $\ell(\tilde{p}_i, \tilde{p}_j)$ to be the *expected travel time* to get from $\tilde{p}_i$ to $\tilde{p}_j$ using solely connections from $\mathcal{C}(\tilde{p}_i, \tilde{p}_j)$. Note that the expected travel time also includes waiting times between subsequent connections. We now use $\tilde{G}$ with ℓ to select important stops by one of the following methods.

Contraction Hierarchies. A fast approach for selecting important super stops is using Contraction Hierarchies [GSSV12]. A contraction routine iteratively removes unimportant vertices from $\tilde{G}$ and adds shortcuts in order to preserve the distances between non-removed vertices. It stops as soon as the number of unremoved vertices is c (an input parameter). It marks the remaining super stops as important, i. e., adds them to $\tilde{\mathcal{S}}_{\text{hub}}$.

Shortest Path Covers. Abraham et al. [ADGW11] observed that Contraction Hierarchies may do a poor job picking the most important vertices in the context of road networks. Hence, they propose using shortest path covers for selecting them. Unfortunately, computing such covers is hard, but the authors propose a polynomial time $\mathcal{O}(\log|V|)$ approximation algorithm which we adapt to our problem by the following approach. It begins with $\tilde{\mathcal{S}}_{\text{hub}} = \emptyset$ and iteratively determines the next most important super stop as the one that covers most (yet uncovered by $\tilde{\mathcal{S}}_{\text{hub}}$) shortest path in $\tilde{G}$. The algorithm stops as soon as it selected c hub super stops. Note that this algorithm requires c times the computation of all-pairs shortest path in $\tilde{G}$. However, $\tilde{G}$ is sufficiently small for this approach to be still practical.

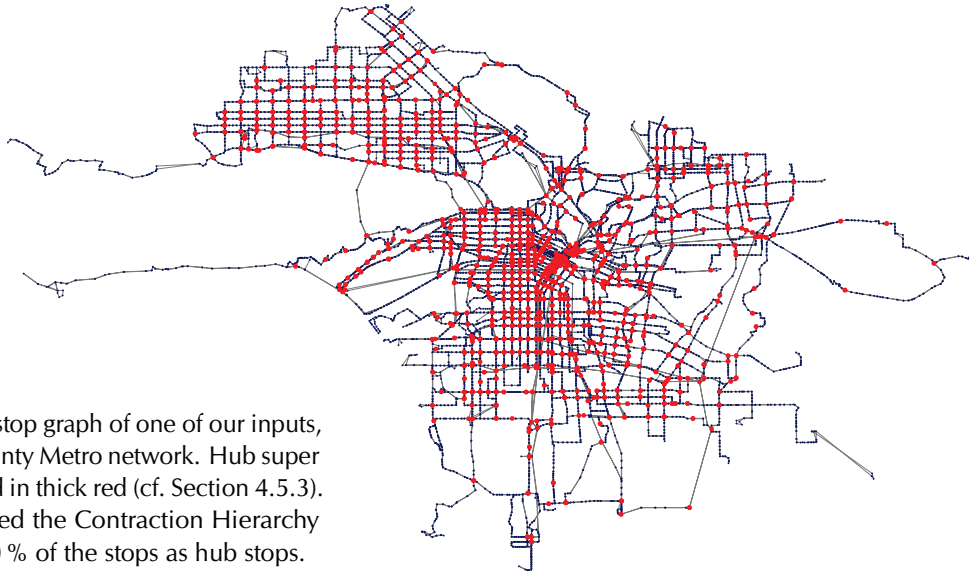


Figure 4.20. Super stop graph of one of our inputs, the Los Angeles County Metro network. Hub super stops are highlighted in thick red (cf. Section 4.5.3). In this figure we used the Contraction Hierarchy method to select 10 % of the stops as hub stops.

4.5.4. Experiments

We conducted experiments on up to 48 cores (4 CPUs, 8 NUMA-nodes, 6 cores per NUMA-node) of an AMD Opteron 6172 machine running SUSE Linux. The machine is clocked at 2.1 GHz, has 256 GiB of RAM, 512 KiB of L2 cache per core, and 6 MiB of L3 cache per NUMA-node. The program was compiled with GCC 4.5, using optimization level 3. Our implementation is written in C++ using the standard template library solely for basic data structures, such as vectors. As parallelization framework we use OpenMP and a 4-heap as priority queue.

To avoid congestion of the memory bus, we keep a copy of the graph in the designated memory area of each NUMA-node.

Inputs. We use three different public transportation networks as inputs: The Los Angeles County Metro (15 146 stops and 979 283 elementary connections), and the complete network of Metropolitan Transport Authority of New York which includes buses, ferries, and subways (16 897 stops and 2 062 846 elementary connections). Moreover, we use the long-distance railway network of Europe. It has 30 517 stations and 1 691 691 elementary connections.

The networks of Los Angeles and New York were created based on the timetable of March 1 2011. The European railway network is based on the timetable of the winter period 1996/1997. Note that the local networks are much denser than the railway network, i. e., the connections per station ratio is significantly higher there. Moreover, our data of the European railway network contains realistic minimum change times for all stations. For the bus networks of New York and Los Angeles this data was not available to us. Hence, we set a minimum change time of 90 seconds for all bus stops.

Table 4.2. Comparison of the realistic time-dependent model to our Coloring Model. We report the number of stops of the timetable, the number of vertices and arcs in the graph, as well as the number of route vertices per stop and the percentage of stops that could be merged (i. e., consisted of only one route vertex).

	Los Angeles		**New York**		**Europe**	
Figure	Routes	Colored	Routes	Colored	Routes	Colored
# Stops	15 146	15 146	16 897	16 897	30 517	30 517
# Vertices	89 111	21 680	79 881	27 203	515 062	83 732
# Arcs	235 394	54 896	198 232	67 105	1 412 082	392 675
Rt. Vertices p. St.	4.9	0.4	3.7	0.6	15.9	1.7
% Merged St.	—	79.5	—	71.7	—	33.2

Footpaths are computed on all networks by our heuristic, see Section 4.3.5.

The timetable data of the local city networks is publicly available via General Transit Data Feeds [Gen10], while the timetable data of the European railway network was kindly given to us by HaCon - Ingenieurgesellschaft [HaC84]. As an example, see Figure 4.20 for the super stop graph of the Los Angeles network.

Modeling

Our first set of experiments focuses on evaluating the models, as presented in Section 4.3. In particular, we compare the realistic time-dependent model with our new Coloring Model. Table 4.2 shows figures on all of our inputs for both models. We observe that using the Coloring Model reduces the graph size for all inputs. The average number of route vertices per stop shrinks by a factor of between 6.1 (New York) and 12.3 (Los Angeles).

Additionally, we observe for many stops that there exists no conflict between any connections. In fact, the model merges the only route vertex with its stop vertex for 79.5 % of the stops in the Los Angeles network. On the other hand, on the European railway network about two thirds of the stops contain more than one route vertex, which stems from the fact that in this network minimum change times are higher, thus, increasing the likelihood of two trains having a conflict.

Since the Coloring Model yields smaller graphs, which improves performance on all our algorithms compared to the realistic time-dependent model, we use the Coloring Model for all subsequent experiments.

One-to-All Queries

Our second set of experiments focuses on the question how well our Parallel Self-Pruning Connection-Setting Algorithm (PSPCS) performs if executed on a varying

number of cores. Therefore, we ran 1 000 one-to-all queries with the source stop picked uniformly at random. We report the average number of connections extracted from the priority queue (sum over all cores) and the average execution time of a query. Table 4.3 reports these figures for a varying number (between one and forty-eight) of cores and different partitioning strategies. In order to evaluate the partitioning, we also report the standard deviation with respect to the execution times of the individual threads. In other words, a low deviation shows a good balance, whereas a high deviation indicates that some threads are often idle.

For comparison, we also report the performance of the label-correcting (LC) approach (cf. Section 4.4.2), as well as of our Connection-Setting Algorithm (CS) without self-pruning enabled. (Think of it as running Dijkstra's algorithm simultaneously for every outgoing connection of the source stop.) Regarding LC, for better comparability, the number of connections figure here indicates the sum of the sizes of the connection points (of the functions) taken from the priority queue.

We observe that our algorithm scales pretty well with increasing number of cores. On both the Los Angeles and New York networks, the number of scanned connections is only increasing mildly with the number of cores. So, on twelve cores we have a speedup factor of around four to eight compared to an execution on one core. On 48 cores, the speedup factor is between 3.6 (Europe) and 17.5 (Los Angeles). The relatively mild speedups on Europe compared to the other networks are explained by the fact that the average number of connections at a station is much smaller than in the dense metropolitan networks. Still, on all cores, we are able to compute all optimal connections for a full day in less than 0.2 seconds. Note that this value is achieved without any preprocessing, hence, we can directly use this approach in a fully dynamic scenario (as discussed, for example, in [FMS08]).

Regarding load balancing, we observe that using the equal number of connections strategy (equiconn) yields (on average) the lowest query times (and deviation). In few occasions, the equal time-slots strategy (equitime) or k-means yield better results, but over all inputs and number of cores, equiconn seems to be the best choice. Hence, we use equiconn as default strategy for all further multi-core experiments. Another—not too surprising—observation is that the deviation increases with increasing number of cores. The more cores we use, the harder a perfect balancing can be achieved.

Comparing our new connection-setting to the label correcting approach, we observe that PSPCS outperforms LC—on Los Angeles and Europe even when PSPCS is executed on a single core. The main reason for this is that the number of connections investigated during execution is much smaller for PSPCS than for LC. On the network of New York, LC is slightly faster than PSPCS on a single core, but already on three cores PSPCS outperforms LC by a factor of two. Note that the number of priority queue operations for LC is up to four times lower than for PSPCS. Hence, the advantage of PSPCS in number of scanned connections does not yield the same speedup in query times.

Table 4.3. One-to-all profile queries with our Parallel Self-Pruning Connection-Setting Algorithm (PSPCS) on varying number of cores cores and different partitioning strategies. We compare PSPCS to the label-correcting approach (LC). The column "Spdup" indicates the speedup in running time of a multi-core over a single-core execution of PSPCS. The column "Dev" reports the standard deviation with respect to the execution times of the individual threads indicating how well the threads are balanced (lower values are better).

	Los Angeles				**New York**				**Europe**			
P	Settl. Conns	Time [ms]	Spd. Up	Dev. [%]	Settl. Conns	Time [ms]	Spd. Up	Dev. [%]	Settl. Conns	Time [ms]	Spd. Up	Dev. [%]
1	844 852	374.0	1.0	—	1 606 515	931.5	1.0	—	550 912	394.9	1.0	—
EQUICONN:												
3	855 676	131.5	2.8	9.1	1 625 545	391.5	2.4	13.9	666 889	162.4	2.4	15.3
6	871 978	72.1	5.2	12.9	1 654 798	165.9	5.6	12.6	843 695	139.5	2.8	18.8
12	904 149	66.1	5.7	20.9	1 711 439	118.1	7.9	16.8	1 172 269	100.9	3.9	15.0
24	967 339	46.4	8.1	22.6	1 822 735	106.9	8.7	20.5	1 709 985	125.8	3.1	21.4
48	1 079 224	21.4	17.5	13.9	2 038 022	57.0	16.3	18.5	2 393 664	109.7	3.6	20.9
EQUITIME:												
3	853 629	153.5	2.4	18.9	1 623 518	384.6	2.4	24.5	651 022	163.7	2.4	17.5
6	865 679	85.6	4.4	25.6	1 645 273	201.0	4.6	26.4	799 641	172.6	2.3	23.4
12	891 822	90.7	4.1	24.9	1 692 424	132.9	7.0	23.7	1 065 354	116.5	3.4	18.2
24	943 625	55.2	6.8	23.4	1 783 835	117.5	7.9	22.2	1 474 137	136.1	2.9	21.4
48	1 022 931	38.2	9.8	21.1	1 953 405	69.7	13.4	19.9	1 970 312	117.3	3.4	21.2
k-MEANS:												
3	852 122	142.2	2.6	17.8	1 619 993	361.8	2.6	22.7	648 190	166.0	2.4	19.1
6	864 301	87.2	4.3	24.5	1 643 853	190.9	4.9	25.1	810 833	113.9	3.5	18.8
12	893 412	89.5	4.2	24.7	1 693 146	171.5	5.4	21.3	1 128 571	118.0	3.3	18.0
24	949 905	44.6	8.4	21.5	1 795 074	92.2	10.1	19.8	1 644 280	122.6	3.2	21.3
48	1 057 201	31.0	12.0	20.8	2 002 726	58.5	15.9	19.0	2 276 361	107.2	3.7	21.8
OTHER ALGORITHMS:												
CS	1 352 894	586.7	—	—	3 327 697	1 965.4	—	—	4 377 790	3 843.3	—	—
LC	2 529 009	445.9	—	—	4 656 646	748.4	—	—	1 278 093	635.3	—	—

When comparing the single core execution of PSPCS to a connection-setting algorithm without self-pruning (CS), we observe that enabling self-pruning makes a significant difference in both scanned connections and running time. Most notably, on Europe the number of connections drops from 4.3 million to 0.5 million together with a drop from 3.8 to 0.4 seconds in running time. The difference is less pronounced on the metropolitan networks, which is due to the fact that these networks inherit a weaker hierarchy, i. e., there are fewer "express" trains (respectively buses) that prune local (slow) trains.

Inter-Thread-Pruning. In our previous experiment (cf. Table 4.3) we did not enable inter-thread-pruning (cf. Section 4.5.2). Hence, in Table 4.4 we compare our Self-Pruning Connection-Setting Algorithm with and without inter-thread-pruning on a varying number of cores P. Thereby, we limit the number of threads we check for a dominating connection to one.

We observe that activating inter-thread-pruning helps reducing the number of scanned connections in all scenarios. Interestingly, even for a sequential execution we are able to reduce the number of scanned connections. Here, the "thread" we check for a dominating connection is the thread itself. By these means, we are able to prune over the boundary of the time period, e. g., for a connection after midnight to prune a connection in the late evening (remember that the timetable in this section is periodic).

While the number of scanned connections decreases with inter-thread-pruning, the additional computational overhead in the algorithm does not always justify the smaller number of scanned connections. Hence, the gain in query time is mostly small. In the network of New York, enabling inter-thread-pruning even leads to slightly worse query times. We conclude that the benefit of inter-thread-pruning is small. Thus, for the sake of simplicity and reduced communication overhead of the algorithm, we disable inter-thread-pruning in subsequent experiments.

Point-to-Point Queries

In this experiment we evaluate our algorithm in a point-to-point scenario. We use all 48 cores as default and evaluate the impact of different distance table sizes. Since these tables need to be precomputed, we also report the preprocessing time and the size of the tables in Megabytes. Furthermore, we report the average number of via super stops per super stop if it were the target of a query. The distance tables are computed by running our parallel one-to-all algorithm on 48 cores from every hub super stop. As strategies for selecting hub stops, we evaluate both the Greedy Covers (GC) and the Contraction Hierarchies (CH) approaches (cf. Section 4.5.3). Table 4.5 gives an overview over the obtained results.

We observe that compared to Table 4.3, the stopping criterion alone (which requires no preprocessing) already accelerates queries by up to 89 % (Europe).

Table 4.4. Comparing our Self-Pruning Connection-Setting Algorithm with and without inter-thread-pruning enabled on a varying number of cores P. The column "Spdup" refers to the speedup in running time over a sequential execution of the same algorithm.

	Without ITP			**With ITP**		
P	Settl. Conns	Time [ms]	Spd. Up	Settl. Conns	Time [ms]	Spd. Up
Los Angeles:						
1	844 852	374.0	1.0	838 331	381.5	1.0
3	855 676	131.5	2.8	836 759	215.9	1.8
6	871 978	72.1	5.2	835 494	72.7	5.2
12	904 149	66.1	5.7	836 186	41.7	9.1
24	967 339	46.4	8.1	856 631	47.6	8.0
48	1 079 224	21.4	17.5	919 060	32.9	11.6
New York:						
1	1 606 515	931.5	1.0	1 595 121	958.3	1.0
3	1 625 545	391.5	2.4	1 594 007	413.8	2.3
6	1 654 798	165.9	5.6	1 594 153	173.9	5.5
12	1 711 439	118.1	7.9	1 600 842	158.0	6.1
24	1 822 735	106.9	8.7	1 625 629	104.7	9.2
48	2 038 022	57.0	16.3	1 711 238	59.5	16.1
Europe:						
1	550 912	394.9	1.0	511 203	373.7	1.0
3	666 889	162.4	2.4	528 588	224.5	1.7
6	843 695	139.5	2.8	610 796	100.2	3.7
12	1 172 269	100.9	3.9	824 653	119.5	3.1
24	1 709 985	125.8	3.1	1 230 380	109.8	3.4
48	2 393 664	109.7	3.6	1 753 982	106.7	3.5

Table 4.5. Performance of our Parallel Self-Pruning Connection-Setting Algorithm (PSPCS) with stopping criterion enabled. As partitioning strategy we use the equal connections method. Moreover, we prune by a distance table as described in Section 4.5.3. The number of hub super stops is given in percentage of input super stops.

	Los Angeles						New York					
	Preprocessing			Queries			Preprocessing			Queries		
Size [%]	Time [m:s]	Space [MiB]	Via St.	Settl. Conns	Time [ms]	Spd. Up	Time [m:s]	Space [MiB]	Via St.	Scnd. Conns	Time [ms]	Spd. Up
0	—	—	—	614 254	19.8	1.0	—	—	—	1 188 870	35.4	1.0
Greedy Covers:												
2.5	2:48	52.5	39.3	392 872	25.7	0.8	5:07	115.0	20.0	547 307	32.3	1.1
5.0	5:15	171.7	8.4	214 620	12.8	1.5	9:57	394.8	4.7	280 011	18.9	1.9
10.0	10:50	577.5	3.7	141 348	10.0	2.0	20:29	1 352.7	2.6	198 315	15.7	2.3
15.0	16:29	1 189.5	2.8	126 509	9.9	2.0	31:45	2 807.8	2.1	181 965	14.8	2.4
20.0	22:24	1 980.7	2.5	121 244	9.8	2.0	43:57	4 791.7	1.9	174 438	14.5	2.4
Contraction Hierarchies:												
2.5	1:09	53.3	295.4	565 832	60.7	0.3	1:43	110.9	165.2	784 445	82.8	0.4
5.0	2:41	196.3	28.0	250 452	26.0	0.8	4:31	412.0	8.1	299 851	12.8	2.8
10.0	6:12	659.0	3.8	128 265	9.8	2.0	10:50	1 500.4	2.6	183 729	10.7	3.3
15.0	9:35	1 323.1	2.7	109 229	8.0	2.5	17:23	3 126.8	1.9	167 777	9.5	3.7
20.0	13:12	2 166.4	2.3	110 502	8.8	2.2	24:53	5 213.8	1.7	162 283	11.9	3.0

	Europe					
	Preprocessing			Queries		
Size [%]	Time [m:s]	Space [MiB]	Via St.	Scnd. Conns	Time [ms]	Spd. Up
0	—	—	—	1 266 720	58.0	1.0
Greedy Covers:						
2.5	44:59	71.9	5.9	347 156	21.5	2.7
5.0	84:13	261.3	3.0	261 894	17.8	3.3
10.0	161:41	930.6	2.2	256 514	18.4	3.2
15.0	216:31	1 956.0	2.1	263 867	19.4	3.0
20.0	280:02	3 354.7	2.0	260 812	18.0	3.2
Contraction Hierarchies:						
2.5	2:32	72.7	42.7	507 466	41.8	1.4
5.0	5:21	269.5	4.9	280 494	19.1	3.0
10.0	12:01	985.8	2.2	220 550	16.1	3.6
15.0	18:37	2 068.6	1.9	208 599	14.1	4.1
20.0	27:01	3 492.4	1.7	218 388	15.4	3.8

When we additionally use a distance table, we can accelerate our queries further. We observe that the size of the distance table has a high impact on the query performance, especially for smaller tables. Augmenting only 2.5 % of the super stops to hub super stops hardly accelerates queries. In fact, especially on the very dense network of Los Angeles, the performance even degrades for small tables, as the average number of required via super stops per target super stop is too high. Note that we need to separate the target super stop by via stops from the network (cf. Section 4.5.3), hence, the more super stops are augmented as hub stops, the less of them are required to separate the target super stop.

On the other hand, augmenting 10 % of the super stops yields additional speedups between 2.0 and 3.6, depending on the input. Larger distance tables hardly pay off: The size of the table increases significantly, and the gain in query performance is little. Hence, selecting 10–15 % of the stops as hub stops seems to be a good compromise.

Regarding the preprocessing effort, we observe that with increasing number of hub stops the size of the tables and the preprocessing time increase as well. Moreover, while the fraction of the preprocessing time spent on selecting hub super stops is negligible when using the Contraction Hierarchies method (CH), it is significant for the Greedy Covers method (GC). This is because for each selected super stop, we need to run an all-pairs shortest-path computation on the (sparse) super stop graph, each of which takes time $O(|\tilde{\mathcal{S}}|^2 \log |\tilde{\mathcal{S}}|)$. Recall that $\tilde{\mathcal{S}}$ is the set of super stops.

However, when using 10 % hub super stops selected by the CH method, we can compute the distance tables in 6 to 10 minutes while the tables consume less than 1.5 GiB space for all of our inputs. For this scenario, we are able to compute all quickest connections on all inputs in less than 16.1 ms time.

A Different Machine

In this final experiment, we run our parallel algorithm on different hardware. Here we use a dual Intel Xeon 5430 machine which has 8 cores on two NUMA-nodes clocked at 2.6 GHz, 32 GiB of RAM and 2 × 1 MiB of L2 cache. To evaluate our algorithm on this machine, we use the one-to-all scenario, however, for the sake of simplicity, only for the equal connections distribution strategy. Table 4.6 shows the obtained results.

We observe that the figures of the sequential algorithms coincide with those in Table 4.3, except that they are scaled: The Xeon machine is slightly faster, since it has a higher clock frequency (2.6 GHz compared to 2.1 GHz of the Opteron machine). Regarding the parallel performance we observe speedups in the range of 3.5 to 5.7 on 8 cores. Again, the number of scanned connections on the networks of Los Angeles and New York is almost independent of the number of cores, even without inter-thread-pruning (which is, again, disabled in this experiment). Concluding, we are able to compute all best connections to all stations in under 131 ms on average in all of our networks on this machine.

Table 4.6. One-to-all profile queries as in Table 4.3, but on an Intel Xeon 5430 machine. Regarding PSPCS, we only report results for the EQUICONN partitioning strategy.

	Los Angeles				New York				Europe			
P	Settl. Conns	Time [ms]	Spd. Up	Dev. [%]	Settl. Conns	Time [ms]	Spd. Up	Dev. [%]	Settl. Conns	Time [ms]	Spd. Up	Dev. [%]
1	844 852	303.5	1.0	—	1 606 515	725.2	1.0	—	550 912	293.6	1.0	—
EQUICONN:												
2	849 553	196.1	1.5	9.5	1 615 576	440.0	1.6	7.4	608 951	166.2	1.8	12.6
4	861 235	92.1	3.3	9.6	1 636 196	224.4	3.2	10.4	726 382	113.4	2.6	15.8
8	882 905	53.3	5.7	13.7	1 674 558	131.0	5.5	11.7	955 412	83.7	3.5	14.2
OTHER ALGORITHMS:												
CS	1 352 894	451.1	—	—	3 327 697	1 363.7	—	—	4 377 790	2 881.5	—	—
LC	2 529 009	356.2	—	—	4 656 646	589.0	—	—	1 278 093	519.3	—	—

4.5.5. Conclusion

In this section, we presented a new parallel algorithm for computing all best journeys of a day from a given stop to all other stops in a public transit network in a single query. To this extent, we exploited the special structure of travel time functions in such networks and the fact that only few connections are useful when traveling sufficiently far away. Introducing the concept of connection-setting, we showed how to transfer the label-setting property of Dijkstra's algorithm to profile queries in transit networks. By the fact that the outgoing connections of the source stop can be distributed to different processors, our algorithm is easy to use in a multicore setup yielding excellent speedups on today's computers. Moreover, utilizing the very same algorithm to precompute connections between important stations, we can greatly accelerate point-to-point queries.

Regarding future work, it will be interesting to incorporate multicriteria connections, e. g., minimizing the number of transfers or incorporating fare zones which is relevant especially in local networks (also see the next section). The main challenge here is to keep up the connection-setting property and to find efficient criteria for self-pruning in such a scenario. Moreover, our algorithm can be seen as a replacement for the Label Correcting algorithm, which is the basis for most of today's time-dependent speedup techniques, e. g., [Del11]. Hence, it would be interested to apply those techniques to our new connection-setting approach.

4.6. Round-Based Public Transit Optimized Router

In this section we introduce RAPTOR, the Round-bAsed Public Transit Optimized Router. It solves the multicriteria problem by computing Pareto sets of journeys optimizing arrival time and the number of transfers taken. Unlike all approaches mentioned so far in this work, it is *not* graph-based. Instead, it directly operates on the data structures of the timetable (such as stops, routes, and trips), thereby, dropping the need for a priority queue. We show that routes may be processed (almost) independently, which allows for an easy parallelization of RAPTOR. Finally, we extend the algorithm to the multicriteria range scenario and also to handle further (arbitrary) criteria (besides arrival time and number of transfers taken).

Overview. The section is organized as follows. First, Section 4.6.1 introduces RAPTOR in its basic variant. Several improvements, such as pruning, are discussed in Section 4.6.2. Section 4.6.3 shows how RAPTOR can be extended to compute *full* Pareto sets using strict domination. In Section 4.6.4 we show how RAPTOR can be easily parallelized. In Section 4.6.5 we discuss exploiting the fact that trips operate with certain frequencies in order to compress the timetable and how RAPTOR can be adapted to handle this compression scheme (FRAPTOR). Section 4.6.6 then extends RAPTOR to handle more criteria (McRAPTOR) and contains details how two important criteria can be incorporated, namely fare zones and reliability of transfers. Multicriteria range queries (rRAPTOR) are covered in Section 4.6.7. Section 4.6.8 reports a detailed experimental study, which also compares RAPTOR to SPCS (cf. Section 4.5), and Section 4.6.9 describes implementation details about the data structures used by RAPTOR. Finally, Section 4.6.10 contains concluding remarks.

References. This section is based on [DPW12a] which appeared at the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12) and [DPW12b] which has been accepted at the INFORMS Journal for Transportation Science. It is joint work with Daniel Delling and Renato F. Werneck. Part of this chapter was developed while the author of this thesis visited Microsoft Research Silicon Valley.

We would also like to thank Dominic Green, Hatay Tuna, Kutay Tuna, and Simon Williams from Microsoft Services UK for inspirational discussions and processing the London transit data.

4.6.1. Basic RAPTOR Algorithm

We now introduce the basic version of RAPTOR. It solves the bicriteria problem minimizing arrival time and number of transfers taken—like LD or MLC. However, our method is not based on Dijkstra's algorithm. In fact, it does not even need a priority queue. Let $p_s \in \mathcal{S}$ be the source stop and $\tau \in \Pi$ the departure time. Recall that our goal is to compute for every number of transfers k a nondominated journey

to a target stop p_t with minimum arrival time having at most k trips. We start with a basic version of the algorithm, then propose some optimizations.

Round-Based Approach. The algorithm works in rounds. Round k computes the fastest way of getting to every stop with at most $k-1$ transfers (i. e., by taking at most k trips). Note that some stops may not be reachable at all. To explain the algorithm, we bound the number of rounds by K (which can be dynamically extended during the algorithm, if necessary). More precisely, the algorithm associates with each stop p a multilabel $(\tau_0(p), \tau_1(p), \ldots, \tau_K(p))$, where $\tau_i(p)$ represents the earliest known arrival time at p with up to i trips. All values in all labels are initialized to ∞. We then set $\tau_0(p_s) = \tau$. We maintain the following invariant: at the beginning of round k (for $k \geq 1$), the first k entries in $\tau(p)$ (from $\tau_0(p)$ to $\tau_{k-1}(p)$) are correct, i. e., entry $\tau_i(p)$ represents the earliest arrival time at p using at most i trips. The remaining entries are set to ∞. The goal of round k is to compute $\tau_k(p)$ for all p. It does so in three stages.

Stage I. The first stage of round k sets $\tau_k(p) = \tau_{k-1}(p)$ for all stops p; this sets an upper bound on the earliest arrival time at p with at most k trips.

Stage II. The second stage then processes each *route* in the timetable exactly once. Consider a route r, and let $\mathcal{T}(r) = (t_0, t_1, \ldots, t_{|\mathcal{T}(r)|-1})$ be the sequence of trips that follow route r, from earliest to latest. When processing route r, we consider journeys where the last (k-th) trip taken is in route r. Recall that $\tau_{\text{ch}}(p_i)$ is the minimum change time at p_i required for changing trips. Let $\text{et}(r, p_i)$ be the earliest trip in route r that one can catch at stop p_i, i. e., the earliest trip t such that $\tau_{\text{dep}}(t, p_i) \geq \tau_{k-1}(p_i) + \tau_{\text{ch}}(p_i)$. Note that (1) this trip may not exist, in which case $\text{et}(r, p_i)$ is undefined, and (2) in the first round we do not need to add the minimum change time $\tau_{\text{ch}}(p_i)$. To process the route, we visit its stops in order until we find a stop p_i such that $\text{et}(r, p_i)$ is defined. This is when we can "hop on" the route. Let the corresponding trip t be the *current trip* for k. We keep traversing the route. For each subsequent stop p_j, we can update $\tau_k(p_j)$ using this trip. To reconstruct the journey, we set a parent pointer to the stop at which t was boarded. Moreover, we may need to update the current trip for k: At each stop p_i along r it may be possible to catch an earlier trip (because a quicker path to p_i has been found in a previous round). Thus, we have to check if $\tau_{k-1}(p_i) + \tau_{\text{ch}}(p_i) < \tau_{\text{dep}}(t, p_i)$ and update t by recomputing $\text{et}(r, p_i)$. Again, we do not need to consider the minimum change time $\tau_{\text{ch}}(p_i)$ in the first round.

Stage III. Finally, the third stage of round k considers footpaths. For each footpath $(p_i, p_j) \in \mathcal{F}$ it sets $\tau_k(p_j) = \min\{\tau_k(p_j), \tau_k(p_i) + \ell(p_i, p_j)\}$. Note that since $\mathcal{F}$ is transitive (see Section 4.1), we always find the fastest walking path, if one exists. The algorithm can be stopped after round k, if no label $\tau_k(p)$ was improved.

Running Time. The worst-case running time of our algorithm can be bounded as follows. In every round, we scan each route $r \in \mathcal{R}$ at most once. If $|r|$ is the number of stops along r, then we look at $\sum_{r \in \mathcal{R}} |r|$ stops in total to process the route. For each stop, we must find the earliest trip $\text{et}(r, \cdot)$. If we keep the list of trips serving r sorted by time, while traversing r we can find all $\text{et}(r, \cdot)$ values with a single sweep over this list, since $\text{et}(r, \cdot)$ can only decrease.

In total, RAPTOR takes $\mathcal{O}\big(K(\sum_{r \in \mathcal{R}} |r| + |\mathcal{T}| + |\mathcal{F}|)\big)$ time, where K is the number of rounds. Note that the running time per round is potentially *sublinear* in the size of the input: The work per route is linear in the number of trips and the size of the route, but most of the departure/arrival times associated with individual trips are not considered. Constant access to the stops along routes and the arrival and departure times of specific trips can be achieved by a few arrays (see Section 4.6.9 for details). In contrast, a similar analysis for the route-based model reveals that MLC and LD are slower by at least a logarithmic factor, due to the priority queues.

4.6.2. Improvements

Having set up the basic version of our algorithm, we now propose some optimizations.

Marking Routes. Iterating over all routes in every round seems wasteful. Indeed, there is no need to traverse routes that cannot be reached by the previous round, since there is no way to "hop on" to any of its trips. More precisely, during round k, it suffices to traverse only routes that contain at least one stop reached with exactly $k-1$ trips. To see why, consider a route whose last improvement happened at round $k' < k-1$. The route was visited again during round $k'+1 < k$, and no stop along the route improved. There is no point in traversing it again until at least one of its stops improves (due to some other route). To implement this version of the algorithm, we *mark* during round $k-1$ the stops p_i for which we improved the arrival time $\tau_{k-1}(p_i)$. At the beginning of round k, we loop through all marked stops to find all routes that contain them. Only routes from the resulting set Q are considered for scanning in round k. Moreover, since the marked stops are exactly those where we potentially "hop on" a trip in round k, we only have to traverse a route beginning at the earliest marked stop it contains. To enable this, while adding routes to Q, we also remember the earliest marked stop in each route. See also Figure 4.21.

Local Pruning. Another useful technique is *local pruning*. For each stop p_i, we keep a value $\tau^*(p_i)$ representing the earliest known arrival time at p_i. Since we are only interested in Pareto-optimal paths, we now only mark a stop during route traversal at round k when the arrival time with k trips is earlier than $\tau^*(p_i)$. Local pruning, thus, allows us to drop the first stage of each round (copying the labels from the previous round): The value $\tau^*(p_i)$ automatically keeps track of the earliest possible time to get to p_i.

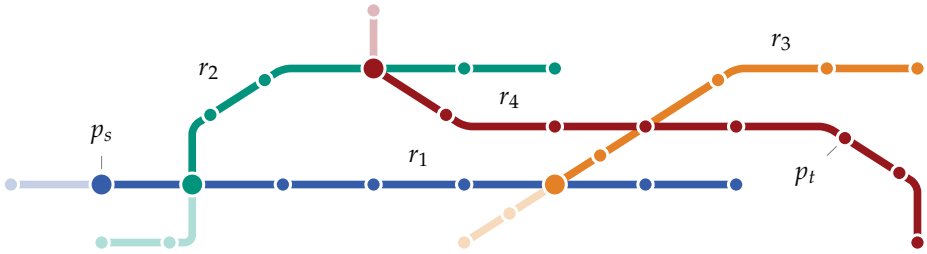


Figure 4.21. Scanning routes for a query from p_s to p_t. Route r_1 is first scanned in round 1, routes r_2 and r_3 in round 2, and finally, route r_4 in round 3. Scanning a route starts at the earliest marked stop (bold). Shallow stops are never visited.

Target Pruning. Note that, as described, RAPTOR computes journeys to *all* stops of the network (which may be useful in some applications). If we are only interested in journeys to a target stop p_t, the performance of RAPTOR can be improved by *target pruning*: During round k, there is no need to mark stops whose arrival times are greater than $\tau^*(p_t)$, which is the best known arrival time at p_t.

Finally, a description in pseudocode of RAPTOR including marking and pruning can be found in Figure 4.22.

4.6.3. Transfer Preferences and Strict Domination

In [BGM10] the authors show that MLC can be extended to the scenario where one is interested in Pareto-optimal solutions with respect to *strict domination*, which means one journey only dominates another if it is strictly better in at least one criterion. This leads to bigger Pareto sets, sometimes called *full* Pareto sets. The motivation for this extension is to output journeys that have transfers at preferred locations. The best journey can be determined in a postprocessing step by looking at all possible combinations of transfer locations.

Transfer Preferences. RAPTOR can handle basic transfer preferences without extending the Pareto set, as follows: When scanning a route r in round k while using trip t, we keep track of the stop (among those where t can be boarded) that maximizes the transfer preference value. Then, whenever we write a label $\tau_k(p)$, we set its parent pointer immediately to the stop with the maximum preference encountered so far.

Strict Domination. In applications that actually require strict dominance, our algorithm can be utilized as follows. Instead of computing all journeys of the Pareto set explicitly (which can be quite many in practice), it outputs a set of *partial trips*, which form a compact representation of the full Pareto set. Thereby, a partial trip $t_{p,p'} = (t, p, p') \in \mathcal{T} \times \mathcal{S} \times \mathcal{S}$ is formally defined as a tuple representing trip $t \in \mathcal{T}$

```
// Input: Source and target stops p_s, p_t and departure time τ.
// Side Effects: Pareto set of journeys for arrival time and number of transfers.

// Initialization of the algorithm
Q ← new Unordered heap of route-stop-tuples
τ_i(p), τ*(p) ← ∞
τ_0(p_s) ← τ
Mark p_s
foreach k ← 1, 2, ... do
    Clear Q
    // Accumulate routes through marked stops from previous round
    forall the marked stop p do
        forall the routes r going through p do
            Let (r', p') ∈ Q where r' = r
            Update Q to contain (r, min_r{p, p'})
        Unmark p
    // Traverse each route
    forall the routes (r, p) ∈ Q do
        t ← ⊥
        foreach stop p_i of r beginning with p do
            // Can the label be improved in this round?
            // Includes local and target pruning
            if t ≠ ⊥ and arr(t, p_i) < min{τ*(p_i), τ*(p_t)} then
                τ_k(p_i) ← τ_arr(t, p_i)
                τ*(p_i) ← min{τ*(p_i), τ_k(p_i)}
                Mark p_i
            // Can we catch an earlier trip here?
            if τ_{k-1}(p_i) ≤ τ_dep(t, p_i) then
                t ← et(r, p_i)
    // Look at footpaths
    forall the marked stops p do
        forall the footpaths (p, p') ∈ F do
            τ_k(p') ← min{τ_k(p'), τ_k(p) + ℓ(p, p')}
            Mark p'
    // Stopping condition
    if no stops are marked then
        stop
```

Figure 4.22. Pseudocode of the RAPTOR algorithm.

restricted to its section from stop p to stop p'. Note that p must be served before p' by the associated route of t.

The algorithm now works as follows. Given a source stop p_s with departure time τ, it first invokes RAPTOR (just as described above) from p_s at departure time τ. This results for every number k of trips at every stop p in an *earliest arrival time* at p when using exactly k trips. To obtain, to a (fixed) target stop p_t the full Pareto set of journeys that have (exactly) k trips, the algorithm invokes the *inverse* variant of RAPTOR from p_t at time $\tau_k(p_t)$, beginning at round k. Inverse RAPTOR just works like regular RAPTOR, except that k is *decremented* in each round, and that it optimizes *latest departure times*. It does so by scanning routes in backward direction. Whenever it visits a stop p of route r, it checks if the departure time $\hat{\tau}_k(p)$ at p (for round k) can be *increased* when departing with the currently considered trip of r. Moreover, it updates the current trip (if possible) to the *latest trip* arriving at p on route r *before* $\hat{\tau}_{k+1}(p)$. The algorithm stops after round 1 is fully executed. This results for every stop p and every round k in a *latest departure time* $\hat{\tau}_k(p)$ for journeys where k trips have already been taken (to get to p from p_s).

Combining both the labels of forward and inverse RAPTOR, we obtain for every stop p and round k an interval $[\tau_k(p), \hat{\tau}_k(p)]$. (We define intervals $[a, b]$ with $a > b$ as the empty interval.) This interval exactly specifies the times at which (a) the k-th trip of any feasible journey (that is contained in the full Pareto set) must arrive at p, and (b), the $(k+1)$-th trip of any journey must depart from p. Therefore, the partial trips that represent the full Pareto set are induced by these intervals. We build them in a quick postprocessing step: For each round k, stop p, and every trip t that departs at p within the interval $[\tau_k(p), \hat{\tau}_k(p)]$, the algorithms traverses t along the stops of its associated route. Then, for each stop p', it check if arriving at p' using trip t is feasible in round $k+1$, i. e., if $\tau_{\text{arr}}(t, p') \in [\tau_{k+1}(p), \hat{\tau}_{k+1}(p)]$ is true. If this is the case, it adds the partial trip $t_{p,p'}$ to the output.

If one is interested in the actual journeys of the full Pareto set, they can be obtained from the partial trips as follows. Consider the directed (acyclic) graph $G = (V, A)$ of *partial trip relations*, which we define as follows. Each vertex $u \in V$ corresponds to a partial trip. An arc (u, v) is added to A, if u ends at the same stop as v begins at, and v stems from the subsequent round of u. Then, exhaustively enumerating all paths from partial trips (vertices) s that begin at p_s in round 0 to partial trips t that end at p_t exactly results in the journeys contained in the full Pareto set.

Note that computing full Pareto sets can be used to compute viable *alternative journeys* between p_s and p_t, even for the *same* values of arrival time and number of transfers. It turns out, that for public transit networks this already results in interesting alternative journeys. To increase the number of alternatives even more, we may further add an extra slack time on the arrival time at p_t: Instead of running inverse RAPTOR from p_t at time $\tau_k(p_t)$, we simply run it at time $\tau_k(p_t) + \epsilon(\tau_k(p_t) - \tau)$, where $\epsilon \geq 0$ is an input parameter which controls the additional amount of slack time.

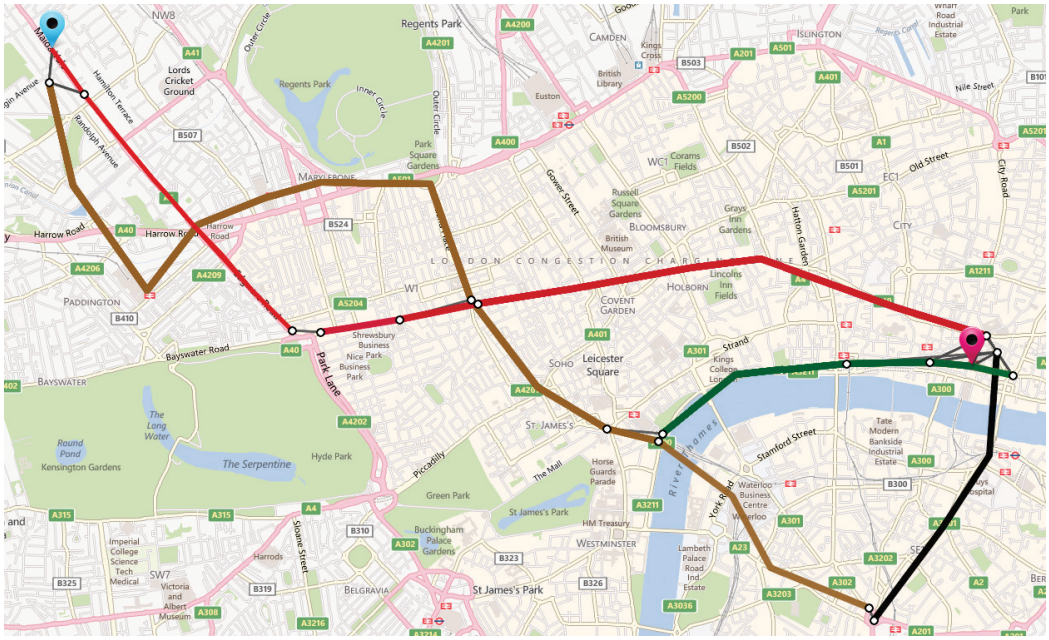


Figure 4.23. Alternative journeys computed by RAPTOR on our London instance (cf. Section 4.6.8). All journeys depart at 6:13 pm at Elgin Avenue/Maida Vale Stn. and arrive at Cannon Street at 6:42 pm with exactly two trips. Any sequence of two trips yields a feasible journey.

Greater values of ϵ lead to higher departure time labels $\hat{\tau}_k(p)$ by inverse RAPTOR, thus, to bigger intervals and greater amount of feasible partial trips. See Figure 4.23 for an example of a full Pareto set computed by RAPTOR. We set $\epsilon = 0.2$ to obtain the journeys in this figure.

4.6.4. Parallelization

While Dijkstra-based algorithms are notoriously hard to parallelize (see e. g. [MS03], [MBBC09]), RAPTOR can be easily extended to work in parallel. Most of the work is spent dealing with individual routes, which are processed in no particular order. If several CPU cores are available, each can handle a different subset of the routes (in each round). During round k, however, multiple threads may attempt to write simultaneously to the same memory location $\tau_k(p)$. Race conditions could be avoided with standard synchronization primitives (such as locks), but that can be costly. Instead, we propose two lock-free parallelization approaches for our algorithm.

Update Logs. If the hardware architecture ensures atomic writes for the values of $\tau_k(p)$, we can just "blindly" write to $\tau_k(p)$, ignoring race conditions. The corresponding memory position will always have a valid upper bound on the arrival time at p, even if a thread could not successfully write a better value. To restore

consistency after the route scanning stage, each thread maintains a log of its update attempts on any value $\tau_k(p)$. At the end of the round, the master thread uses the logs to correct the labels sequentially. The same technique can also be used to keep $\tau^*(p)$ consistent. We call this approach *update log parallelization*.

Conflict Graphs. If atomic writes are not guaranteed, we can still avoid locks with the *conflict graph approach*. We use the fact that any two routes that have no stop in common can be safely scanned in parallel. In a quick preprocessing step, we build an undirected *conflict graph* G, where vertices correspond to routes and there are edges between any two routes that share at least one stop. We then greedily color the routes such that no two adjacent routes share the same color. Routes with the same color can always be processed independently.

To implement this approach efficiently, we order the routes according to their colors (with ties broken arbitrarily) to obtain a sequence $\mathcal{R} = \{r_0, r_1, \ldots, r_{|\mathcal{R}|-1}\}$. We then compute for every route r_i a *dependent route* $\text{pre}(r_i) = r_j$, defined as the highest-indexed conflicting route that appears before i in the order ($j < i$). The route scanning stage is now modified as follows. When a core becomes available, it is assigned the next (in index order) available unprocessed route r_i and waits (in a busy loop) until *all* routes up to $\text{pre}(r_i)$ have been fully processed. Once this happens, it can safely process r_i: Conflicting routes r_j with $j < i$ have already been processed, and those with $j > i$ will wait until r_i is finished. Threads can use shared memory to communicate to others that their own routes have been processed, ensuring no two threads ever write to the same location. Unmarked routes can be skipped and set to processed. In dynamic scenarios, route dependencies must be updated whenever a route changes, but this takes negligible time (below a second).

4.6.5. Timetable Compression

Realistic timetables are often (partially) periodic: Trips of the same route operate with fixed frequencies over certain timespans during the day. For example, a bus line operates every ten minutes from 6 am to 4 pm and every 15 minutes from 4 pm to 9 pm. Up to now, we stored each individual trip explicitly. To save memory, we exploit such periodicities to compress our data structures in a quick preprocessing step.

For each route r we consider its trips in order, from earliest to latest, and group contiguous sequences of trips if and only if each pair of subsequent trips (in the group) shares the same departure/arrival time interval at *every* stop along the route.

More formally, for subsequent trips t_i, t_{i+1} and t_j, t_{j+1} (in the group) we require

$$\tau_{\text{arr}}(t_{i+1}, p) - \tau_{\text{arr}}(t_i, p), \tag{4.15a}$$
$$\tau_{\text{arr}}(t_{j+1}, p) - \tau_{\text{arr}}(t_j, p), \tag{4.15b}$$
$$\tau_{\text{dep}}(t_{i+1}, p) - \tau_{\text{dep}}(t_i, p), \text{ and} \tag{4.15c}$$
$$\tau_{\text{dep}}(t_{j+1}, p) - \tau_{\text{dep}}(t_j, p) \tag{4.15d}$$

to be equal among all stops p of the route r. We refer to this time interval as the *periodicity* of the trip group.

We now delete all but the first (i. e., earliest) trip t in each group and additionally store with t its periodicity $\text{per}(t)$ and the size of its group as $\text{size}(t)$. Note that for aperiodic trips we have $\text{size}(t) = 1$, and $\text{per}(t)$ is undefined.

To make use of the compressed timetable, we modify RAPTOR to expand trips on the fly. When processing a route r, we not only keep track of the current trip t, but also of an *offset* $0 \leq \alpha < \text{size}(t)$. Whenever we evaluate the departure time of t at a stop p, we compute $\tau_{\text{dep}}(t, p) + \alpha \cdot \text{per}(t)$ (the arrival time is computed analogously). Accordingly, $\text{et}(r, p)$ now returns a trip/offset pair (t, α). We refer to RAPTOR on a frequency-compressed timetable as FRAPTOR.

4.6.6. More Criteria: McRAPTOR

In this section we show how RAPTOR can be extended to handle additional criteria, such as fare zones and reliability. We call the resulting algorithm McRAPTOR (for "more criteria RAPTOR"). For the special case of bicriteria range queries, Section 4.6.7 will present a tailored extension, which we call rRAPTOR.

Extending RAPTOR. Recall that plain RAPTOR stores exactly one value $\tau_k(p)$ per stop and round. To extend the algorithm to more criteria, we keep multiple nondominating labels for each stop p in round k, similarly to MLC (cf. Section 4.4.2). We store these labels in *bags*, denoted by $B_k(p)$.

The algorithm is then modified as follows. When processing a route r, we first create an empty *route bag* B_r which keeps track of all good journeys whose last trip is in route r. Therefore, each label L in the route bag has an associated active trip $t(L)$. When traversing the stops of r in order, we process each stop p in three steps.

The first step updates the arrival times of every label $L \in B_r$ to the arrival times of their associated trips $t(L)$ at p. Note that if two labels have the same associated trip, one might be eliminated.

In the second step, we *merge* B_r into $B_k(p)$ by copying all labels from B_r to $B_k(p)$ and discarding dominated labels in $B_k(p)$.

The final step merges $B_{k-1}(p)$ into B_r and assigns trips to all newly-added labels.

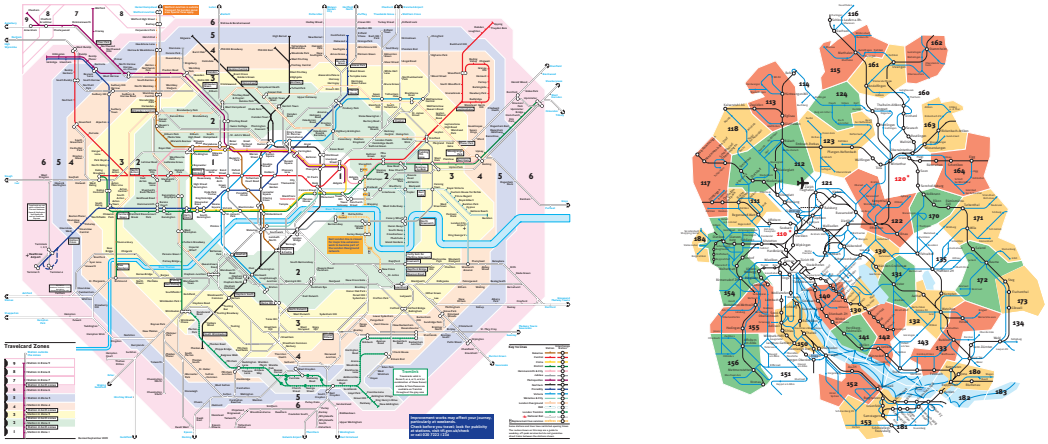


Figure 4.24. Two examples of fare zones: London, UK from 2012 [Tra00] (left) and Zürich, Switzerland from 2012 [Zü90] (right).

The footpaths stage of the algorithm is also modified. When looking at a footpath (p_i, p_j), we create a temporary copy of $B_k(p_i)$ and add $\ell(p_i, p_j)$ to the arrival time of every label. Then we merge this bag into $B_k(p_j)$.

Local and Target Pruning. We also adapt local and target pruning. Similarly to τ^* in RAPTOR, we keep for every stop p a *best bag* $B^*(p)$ that represents the nondominated set of labels over all previous rounds. Thus, whenever we are about to add a label L to a bag $B_k(p)$, we check if L is dominated by $B^*(p)$ or $B^*(p_t)$ (recall that p_t is the target stop). If either is the case, L is not added to $B_k(p)$. Otherwise, we also update $B^*(p)$ by adding L to $B^*(p)$, if necessary. See Section 4.6.9 for details on the implementation of bags.

Parallelization. Like RAPTOR, McRAPTOR scans routes in no particular order and, thus, can be parallelized in the same way. However, since updates to $B_r(p)$ cannot be atomic, we must use the conflict graph approach.

First Example: Fare Zones

We now consider a practical scenario: fare zones. Transit agencies often assign each stop p to one (or multiple) fare zones from a set $\mathcal{Z}$ (see Figure 4.24). The price of a journey is then determined by which fare zones it touches. Since it is often not clear how to handle prices directly during the algorithm, it is simpler to keep track of fare zones instead. Thus, we are interested in computing all Pareto-optimal journeys including the set of touched fare zones as a criterion. Precise fare information can then be determined in a (quick) postprocessing step.

Incorporating Fare Zones. We handle this scenario as follows. Each label is a tuple $L = (\tau(L), z(L))$, where $z(L) \subseteq \mathcal{Z}$ is the set of touched fare zones so far. (Recall that number of transfers are not part of the label since they are handled implicitly by RAPTOR.) Here, a label L_1 dominates L_2 if and only if $\tau(L_1) \leq \tau(L_2)$ and $z(L_1) \subseteq z(L_2)$. Note that $z(p)$ is a cost imposed by *stops* rather than travel. We initialize the source bag $B_0(p_s)$ with a label $(\tau, z(p_s))$. Moreover, each time we are about to merge a label L into a bag $B_k(p)$, we first update $z(L) \leftarrow z(L) \cup z(p)$. To implement $z(\cdot)$ efficiently, we use integers as bit sets (one bit per fare zone). Domination is tested by bitwise-and, and set union is equivalent to bitwise-or.

Second Example: Reliability

Another practical scenario we consider is the reliability of transfers. Following Disser et al. [DMS08], we consider the reliability of one transfer (i. e., change of trips) as a function of the *buffer time* of a transfer to the interval $[0, 1]$. Here, the buffer time is the time difference between departure and arrival of two subsequent trips $t_1 \neq t_2$ of a journey at some stop. It represents the maximum time t_1 may be delayed before t_2 is missed. The reliability of a transfer therefore represents the probability that the transfer will be made successfully. The reliability of a journey is the product over the reliabilities of all its transfers.

Our experiments consider two natural reliability functions. The first is a (piecewise) linear function

$$\text{rel}\colon \tau \mapsto \min(a \cdot \tau + b, 1), \tag{4.16}$$

the second a discretized exponential function

$$\text{rel}\colon \tau \mapsto 1 - e^{\ln(1-a) - b/\tau} \tag{4.17}$$

which has been proposed by Disser et al. [DMS08]). We may use a and b to set the reliability value for a buffer time of 0 as well as the buffer time for which the reliability reaches a "sufficiently high" value, e. g., 0.99.

To limit the number of Pareto-optimal journeys, we may—like [DMS08]—further discretize the interval $[0, 1]$ by subdividing it into a fixed number (e. g., 10) of equivalence classes of equal width.

Incorporating Reliability. We incorporate this criterion into McRAPTOR as follows. Each label is a tuple $L = (\tau(L), x(L))$, where $x(L) \in [0, 1]$ is the reliability value so far. A label L_1 dominates L_2 iff $\tau(L_1) \leq \tau(L_2)$ and $x(L_1) \geq x(L_2)$. The source bag $B_0(p_s)$ is initialized with a label $(\tau, 1)$. Moreover, we modify the third stage of every round (where we assign trips). Each label $(\tau(L), x(L)) \in B_{k-1}(p)$ may now result in *several* new labels (with assigned trips) in B_r: Taking a *later* trip (of r) in favor of a *higher* reliability may contribute to a Pareto-optimal solution.

Hence, in a loop we create new labels $(\tau(L'), x(L'))$ for each trip t that can be caught after $\tau(L)$ (ordered from earliest to latest). We set $x(L') = x(L) \cdot \text{rel}(\tau_{\text{dep}}(t, p) - \tau(L))$, possibly discretizing the value. We may stop the loop as soon as $\text{rel}(\tau_{\text{dep}}(t, p) - \tau(L))$ reaches 1 (which we ensure by discretizing rel accordingly). Each newly created label L' is then merged into B_r, thereby discarding dominated labels.

Note that incorporating reliability into MLC also requires modifications, which are similar to McRAPTOR: One label may result in several new labels when evaluating arcs of the graph. Additionally, one must use a weaker domination rule at route vertices (cf. Section 4.3.3). Consider a vertex representing a route r. It may have two types of labels: *Transfer* labels have the corresponding stop vertex as their parent, while *route* labels have another route vertex as parent. A transfer label can only dominate a route label if and only if their respective arrival times are more than the maximum buffer time (of the reliability function) apart. The reason for this is that the reliability for a transfer into route r will only be considered when relaxing the next arc on route r.

4.6.7. Range Queries: rRAPTOR

As explained in Section 4.2, we can implement range queries using McRAPTOR by simply adding departure times as a criterion to the labels. In practice, however, we obtain a faster algorithm by extending RAPTOR using techniques from Section 4.5 in the context of SPCS. In particular, it does not use costly bags. The resulting algorithm is called rRAPTOR.

Let $\Delta \subseteq \Pi$ be the input time range. First, we accumulate into a set Ψ all departure times of trips t at the source stop p_s that depart within Δ. We then run standard RAPTOR for every departure time $\tau \in \Psi$ independently. This results in a label $\tau_k(p)$ for every stop p, departure time τ, and round k. However, not all journeys from Ψ are useful to get to p. More precisely, a journey J_1 dominates a journey J_2 if and only if $\tau_{\text{dep}}(J_1) \geq \tau_{\text{dep}}(J_2)$ and $\tau_{\text{arr}}(J_1) \leq \tau_{\text{arr}}(J_2)$.

Integrating Domination. To integrate this domination rule, we order Ψ from latest to earliest and then run RAPTOR for every $\tau \in \Psi$ in order, but we keep the labels $\tau_k(p)$ between rounds, instead of reinitializing them. To see why this is correct, note that $\tau_k(p)$ corresponds to an intermediate journey departing from p_s *no earlier* than journeys computed in the current run (recall that Ψ is ordered). Thus, if $\tau_k(p)$ is smaller, we also know how to reach p *earlier*. Hence, we can safely prune the current journey. However, we cannot use local pruning, since the best arrival times $\tau^*(p)$ do not carry over to earlier departures. Instead, at the beginning of round k we set $\tau_k(p) = \tau_{k-1}(p)$ for all stops where $\tau_{k-1}(p)$ improves $\tau_k(p)$.

RAPTOR's parallelization techniques also work for rRAPTOR. However, since $|\Psi|$ is usually larger than the number P of CPU cores, in practice we use the same approach as in Section 4.5.2. We first partition Ψ into contiguous subsets $\Psi_0, \ldots, \Psi_{P-1}$ of equal

Table 4.7. Size figures for our input instances. The graph figures refer to the realistic time-dependent model graph (cf. Section 4.3.3).

Figure	London	Los Angeles	New York	Germany	Europe
Stops	20 843	15 003	17 894	6 822	30 517
Routes	2 240	1 099	1 393	9 348	44 751
Trips	133 011	16 376	45 299	104 560	899 485
Trips (Compressed)	52 386	10 554	20 031	48 377	443 435
Foot Paths	45 652	15 482	49 858	0	0
Departure Events	5 130 905	931 846	1 825 129	1 019 830	8 341 980
Min. Change Times	no	no	no	yes	yes
Vertices (Graph)	101 524	81 657	66 124	118 365	550 654
Arcs (Graph)	285 510	214 369	193 159	325 292	1 516 012

size. Then each core i runs rRAPTOR on Ψ_i independently. The results are merged in the end, and dominated journeys are discarded.

4.6.8. Experiments

In this section we present an experimental study to evaluate our RAPTOR algorithms. Our main benchmark input uses realistic data from Transport for London [Tra00]. It includes tube (subway), buses, tram, Dockland Light Rail (DLR), and ferries. We extracted a Tuesday from the periodic summer schedule of 2011, which is publicly available from the London Data Store [Lon11]. The network has 20 843 stops, 2 240 routes served by 133 011 trips, and a total of 5 130 905 distinct departure events (a trip departing from a stop). Moreover, there are 45 652 footpaths in the network. Figure 4.3 on Page 56 shows the stop graph of this instance. When applying timetable compression (cf. Section 4.6.5), the number of trips is reduced to 52 386 (a factor of 2.5). Each tube and DLR station is also assigned to one of 11 fare zones. In London a tube ticket automatically includes unlimited bus rides. Thus, we assign bus stops to a special fare zone that every tube/DLR station is also a member of. We compare our algorithms to existing graph-based techniques and also SPCS. They all use the realistic time-dependent model graph (cf. Section 4.3.3), which has 100 524 vertices and 285 510 arcs. These figures are also shown in Table 4.7, together with the corresponding sizes for the other instances we consider, which we discuss later in this Section.

All experiments were conducted on a dual 8-core Intel Xeon E5-2670 machine that has 16 cores in total, clocked at 2.6 GHz, with 64 GiB of DDR3-1600 RAM. We disabled hypterthreading for all experiments. We implemented all algorithms in C++ (with OpenMP for parallelization), and used GCC 4.6.2 (64 bit) with full optimization as compiler. To evaluate performance, we ran 10 000 queries with source/target

Table 4.8. Evaluation of different variants of RAPTOR and FRAPTOR on the London instance, compared to Time-Dijkstra (TD), Layered Dijkstra (LD), and Multi-Label-Correcting (MLC). Bullets (•) indicate different features: minimize arrival time (Arr), minimize number of transfers (Tran), target pruning (Prn), unpacking journeys (Unp).

Algorithm	Arr.	Tran.	Prn.	Unp.	# Rnd.	# Relax. p. Route	# Visits p. Stop	# Comp. p. Stop	# Jn.	Time [ms]
RAPTOR	•	•	○	○	9.8	4.0	15.2	12.3	1.8	7.6
RAPTOR	•	•	•	○	8.3	3.0	10.6	10.9	1.8	5.4
RAPTOR	•	•	•	•	8.3	3.0	10.6	10.9	1.8	6.6
FRAPTOR	•	•	•	○	8.3	3.0	10.6	10.9	1.8	5.6
TD	•	○	•	○	—	—	2.6	7.4	0.9	11.0
LD	•	•	•	○	—	—	7.1	15.6	1.8	28.7
MLC	•	•	•	○	—	—	6.0	23.7	1.8	50.0

stops and departure time selected uniformly at random. Results for more realistic distributions are similar.

RAPTOR. In our first set of experiments we evaluate RAPTOR (cf. Section 4.6.1) and compare it to LD and MLC (cf. Section 4.4.2), which solve the same problem (finding Pareto sets according to arrival times and number of transfers). With RAPTOR we separately evaluate the impact of (1) target pruning, and (2) the overhead of tracking paths to output full journey descriptions. Additionally to RAPTOR, we also evaluate FRAPTOR, which uses timetable compression (cf. Section 4.6.5). Note that we always make use of marking and local pruning.

All other algorithms are fully optimized: LD has pruning enabled and MLC uses pruning, label-forwarding, and hop-avoidance. For comparison, we also report the performance of Time-Dijkstra (TD), which solves the (simpler) earliest arrival problem, which does not take the number of transfers into account.

The results are presented in Table 4.8. We report the average number of visits and label comparisons per stop, the average size of the Pareto sets (number of journeys) output, and the average (sequential) running time in milliseconds. Moreover, for RAPTOR we report the average numbers of rounds, as well as the average number of times each route is processed.

We observe that, on average, RAPTOR (without target pruning) performs 9.8 rounds before it can stop (i. e., no labels can be improved) and scans each route 4 times. Recall that without target pruning, we compute optimal journeys to *all* stops in the network (which may be useful in some applications). Enabling target pruning reduces the number of rounds to 8.3 with 3 route scans on average. FRAPTOR on the compressed timetable is only 4 % slower than RAPTOR, which is due to expanding trips on the fly. However, it keeps 2.5 times fewer trips (with their associated

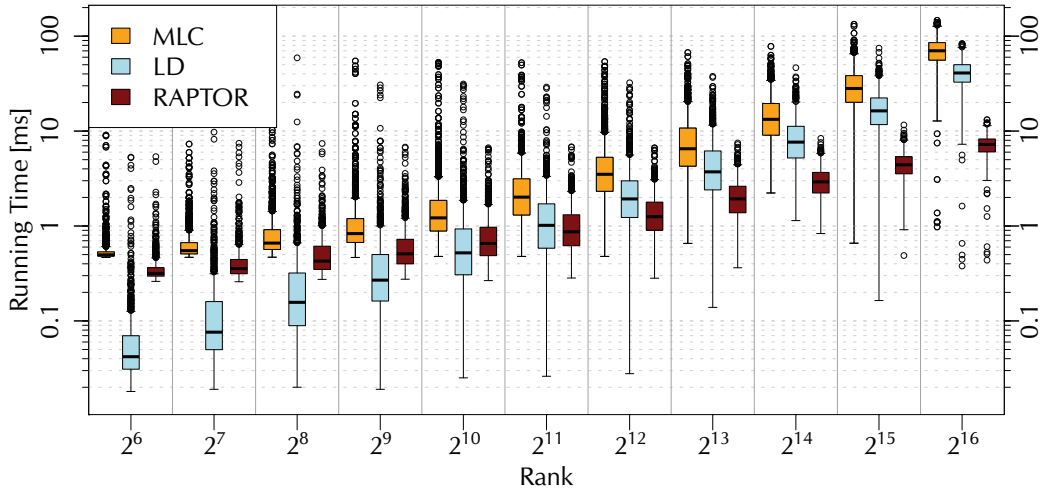


Figure 4.25. Running time of MLC, LD, and RAPTOR on the London instance subject to the Time-Dijkstra Rank. Smaller ranks indicate more local queries.

departure/arrival times) in memory.

When considering the number of label comparisons per stop, we see that RAPTOR, MLC, and LD are no more than a factor of 2 apart. However, RAPTOR strongly benefits from its simpler data structures, better locality, and lack of a priority queue: With an average query time of 5.4 ms, it is 9 times faster than MLC, and 5 times faster than LD. Even TD, which only minimizes arrival time (regardless of the number of transfers), is outperformed by RAPTOR: It outputs half the number of journeys in twice the amount of time. Although TD could be accelerated using models yielding smaller graphs (as in [BDGM09, DKP12, Gei10]), [BDGM09] show that these models would make multicriteria queries more complicated.

When we are also interested in unpacking full journey descriptions, we observe that managing parent pointers within RAPTOR increases the running time by 22 % to 6.6 ms on average.

Local Queries. We also evaluate local queries with RAPTOR using the Dijkstra rank method, introduced in [SS05] in the context of road networks. In our scenario, we determine ranks by running Time-Dijkstra queries without stopping criterion from 10 000 source stops p_s (with random departure times). For each 2^i-th (for integral i) vertex extracted from the priority queue, we look up its corresponding stop p_t and create a p_s–p_t query. These queries are then run with RAPTOR in random order. Figure 4.25 presents results using a box and whiskers plot. Besides RAPTOR, we also evaluate MLC and LD.

We observe that MLC consistently performs worse than both RAPTOR and LD, due to its complicated handling of bags. Interestingly, LD is up to an order of magnitude

Table 4.9. Comparing several extensions of RAPTOR on the London instance (see Sections 4.6.6 and 4.6.7). We also include the Multi-Label-Correcting (MLC) and Self-Pruning Connection-Setting (SPCS) algorithms. Besides arrival time (Arr), the criteria we may consider are number of transfers (Tran), range (Rng), fare zones (Fare), and reliability (Rel).

Algorithm	Arr.	Tran.	Rng.	Fare	Rel.	# Rnd.	# Relax. p. Route	# Visits p. Stop	# Comp. p. Stop	# Jn.	Time [ms]
McRAPTOR	●	●	●	○	○	9.4	3.8	14.1	1 056.4	15.9	219.9
rRAPTOR	●	●	●	○	○	139.0	36.5	119.0	110.2	15.9	61.3
SPCS	●	○	●	○	○	—	—	31.7	87.1	7.4	177.1
McRAPTOR	●	●	○	●	○	10.6	4.5	16.4	277.5	8.8	100.9
MLC	●	●	○	●	○	—	—	22.7	818.2	8.8	304.2
McRAPTOR	●	●	○	○	●	8.4	3.1	11.1	89.6	4.7	71.9
MLC	●	●	○	○	●	—	—	17.3	286.6	4.7	239.8

faster than RAPTOR for very local queries (rank below 2^{11}). The reason for this is that RAPTOR must process routes in full length to ensure correctness (even when the source and target stops are close). For higher ranks, RAPTOR outperforms LD (and MLC) by up to an order of magnitude.

Extensions of RAPTOR. Next, we evaluate both McRAPTOR and rRAPTOR (cf. Sections 4.6.6 and 4.6.7). For rRAPTOR, we fix the time range to 2 hours (other time ranges are evaluated later), and for McRAPTOR, we consider three variants. The first emulates a two-hour range query by using departure time as an additional criterion, the second uses fare zones, and the third uses reliability (as discussed in Section 4.6.6).

Fare zones are implemented using bit sets. For reliability, we use the exponential function mentioned in Section 4.6.6, given by

$$\mathrm{rel}\colon \tau \mapsto 1 - e^{\ln(1-a)-b/\tau}. \tag{4.18}$$

We set a and b such that $\mathrm{rel}(0\,\mathrm{min}) = 0.5$ and $\mathrm{rel}(10\,\mathrm{min}) = 0.99$, subdividing the codomain of rel into 10 equivalence classes of equal width. Also, we store all relevant values of rel (in the range $[0\,\mathrm{min}, 10\,\mathrm{min}]$) into a lookup table, which accelerates the evaluation of rel during queries.

We compare our algorithms to two versions of MLC: one optimizes arrival time, transfers, and fare zones, and the other arrival time, transfers, and reliability. We also compare our algorithm to SPCS from Section 4.5 (with a 2-hour range). Recall that SPCS is a range query minimizing only arrival time (regardless of transfers). The results are presented in Table 4.9. Note that columns Arr (arrival time), Rng (range), Tran (transfers), Fare (fare zones), Rel (reliability) indicate which criteria each method takes into account.

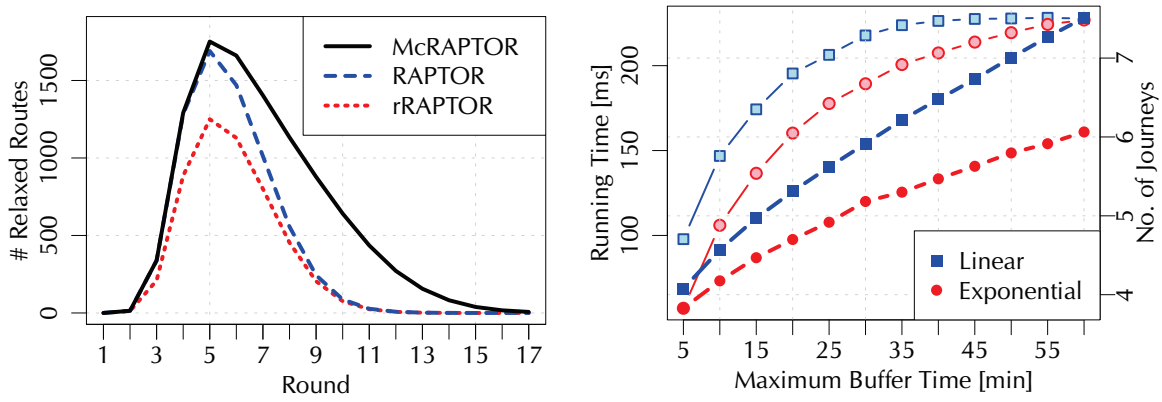


Figure 4.26. Left: Number of relaxed routes per round. For rRAPTOR, we normalize the plot by the number of calls to RAPTOR within each query. Right: Running time (thick) and number of journeys (thin) of McRAPTOR (with criteria arrival time, transfers, and reliability) when varying the maximum buffer time of the reliability function.

Recall that rRAPTOR runs RAPTOR repeatedly (without reinitializing labels). In this experiment, it actually does so 20.7 times on average. Its performance reflects this: It runs 17 times as many rounds and takes 61 ms on average. Using McRAPTOR to emulate the same range queries reduces the number of rounds (relative to rRAPTOR), but running times more than triple. Again, we profit from the simpler data structures. McRAPTOR handles bags of labels instead of running more rounds, which is costly. Compared to pure RAPTOR, taking London's fare zones into account results in 4.9 times more reported journeys. Using McRAPTOR, we achieve a running time of 101 ms, a factor of 3 faster than MLC. Using reliability with McRAPTOR yields similar figures: We output 2.6 times the number of journeys (compared to RAPTOR) in 72 ms. Note that McRAPTOR's speedup over MLC is less than the factor of 9 for RAPTOR (cf. Table 4.8); unlike RAPTOR, McRAPTOR also uses costly bags.

Number of Rounds. Figure 4.26 (left) shows the number of scanned routes per round for RAPTOR, rRAPTOR, and McRAPTOR. We normalize rRAPTOR's plot by the number of calls to RAPTOR within each query, i. e., we report the average number of routes visited in each call to RAPTOR. All algorithms reach the entire network within about 5 rounds, when most routes are scanned. Beyond that, fewer routes are useful, and the algorithms begin running dry. McRAPTOR takes longer to converge, while rRAPTOR generally scans less routes (per departure time) than RAPTOR, since it can prune across different departure times.

Impact of Reliability. Figure 4.26 (right) presents the performance of McRAPTOR when varying the reliability function (with arrival times and number of transfers as other criteria). We compare the linear function to the exponential function (cf. Sec-

Table 4.10. Parallel performance of the RAPTOR, McRAPTOR, rRAPTOR, and SPCS algorithms.

					1 core		4 cores		8 cores		16 cores	
Algorithm	Arr.	Tran.	Rng.	Fare	# Comp. p. Stop	Time [ms]	# Comp. p. Stop	Time [ms]	# Comp. p. Stop	Time [ms]	# Comp. p. Stop	Time [ms]
RAPTOR	●	●	○	○	10.9	5.4	11.0	3.0	11.0	2.6	11.1	3.1
rRAPTOR	●	●	●	○	110.2	61.2	119.9	21.3	132.5	16.5	154.6	17.8
McRAPTOR	●	●	●	○	1 099.4	231.9	1 099.4	72.3	1 099.9	46.8	1 100.3	53.3
McRAPTOR	●	●	○	●	290.8	109.9	290.6	37.8	290.5	25.5	290.7	29.6
SPCS	●	○	●	○	87.1	176.9	101.7	57.3	120.1	45.0	149.8	38.9

tion 4.6.6). We fix $\mathrm{rel}(0\,\mathrm{min}) = 0.5$ and vary the maximum buffer time τ_m (for which $\mathrm{rel}(\tau_m) = 0.99$ is reached) from 5 to 60 minutes. Again, we use 10 equivalence classes of equal width to subdivide the codomain of rel (cf. Section 4.6.6). We plot the average running time over 1 000 queries for each value of τ_m.

We see that the running time increases from around 60 ms (for $\tau_m = 5\,\mathrm{min}$) to almost 250 ms (for $\tau_m = 60\,\mathrm{min}$): Higher maximum buffer times yield more Pareto-optimal journeys and, therefore, more work for McRAPTOR. The exponential function always has faster running times compared to the linear function. While the linear function distributes the buffer times (in the range from 0 to τ_m) evenly among the equivalence classes, the exponential function maps values to equivalence classes of high reliability earlier, thus reducing the number of Pareto-optimal solutions.

Parallelization. Table 4.10 shows the parallel performance of our algorithms. Since writes to the labels $\tau_k(p)$ are atomic for RAPTOR, we use update logs; McRAPTOR is parallelized using conflict graphs.

Among the Dijkstra-based algorithms, only SPCS can be parallelized efficiently across departure times. We ran each algorithm on one, four, eight, and 16 cores, pinning thread i to core i. Note that by this configuration, we utilize only one of the two CPUs in our machine for up to eight cores.

Comparing the single-core execution of the parallel implementations (see Table 4.10) with the sequential ones (see Tables 4.8 and 4.9), we observe a slowdown of less than 10 % for all algorithms. Some slowdown is expected because we introduce additional work for our parallel implementations (see Section 4.6.4). On eight cores, RAPTOR achieves a speedup of only 2.1. Recall that we only parallelize scanning routes, which limits the speedup due to Amdahl's Law (see [Amd67]).

Because McRAPTOR spends more time on each route (due to the costly processing of bags), it benefits more from parallelization (a factor of up to 5 with fare zones or range query emulation). Finally, rRAPTOR achieves a speedup of 3.7 on eight cores, which is consistent with SPCS. Using 16 cores hardly pays off. Compared to

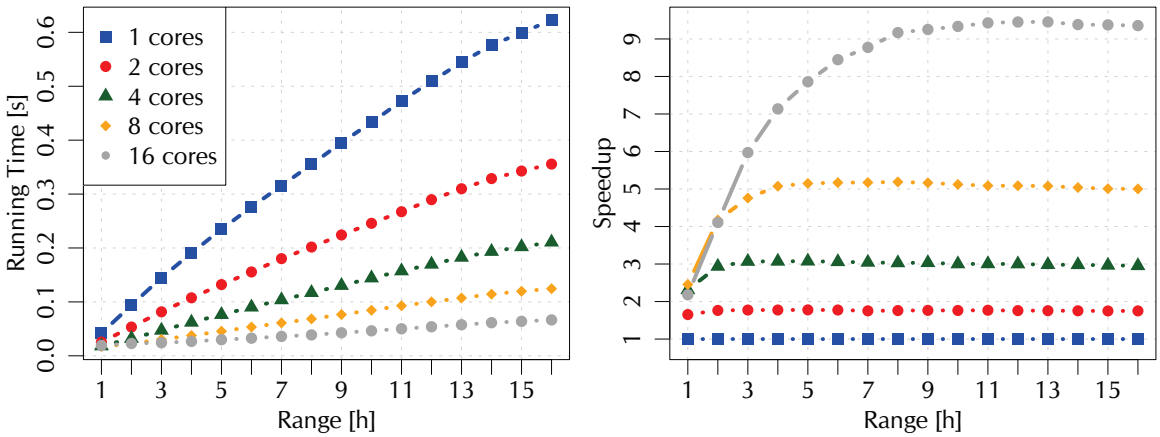


Figure 4.27. Evaluating rRAPTOR with varying range. All queries have a fixed departure time at 6:00 am. The key of the speedup plot (right) corresponds to that of the running time plot (left).

eight cores, for most of the algorithm the performance even gets slightly worse again. Increased memory contention is a factor in this case.

Impact of Range. Figure 4.27 evaluates our parallel implementation of rRAPTOR for varying range values: We fix the departure time to 6:00 am, but vary the range from 1 to 16 hours. For each range value we run 1 000 queries between (the same) random pairs of stops. We report the average running time on one, two, four, eight, and 16 cores, using the same configuration as above.

We observe that increasing the range results in higher running times, with (almost) linear correlation. Note that journeys departing late (at night) might not always reach the target stop due to London's hours of operation. This may in particular occur for high range values, which explains the slight sublinear growth rate for ranges greater than 12 hours. As Figure 4.27 shows, parallelizing rRAPTOR pays off in all cases: For a range of 16 hours we achieve speedup factors of 5.1 on eight, and 9.4 on 16 cores over the sequential algorithm. Using eight cores, we can compute multicriteria range queries in less than 0.15 sec in all cases.

Additional Inputs. We now consider four more test inputs: Los Angeles and New York, which are also metropolitan networks, and two railway networks, Germany and Europe. We generated the local networks from publicly available feeds using the [Gen10]. The railway data was kindly provided to us by HaCon. Los Angeles and New York contain subways and buses, and in both cases we use an extract of August 10, 2011 (a Wednesday) to create the timetable. The German network is based on the winter schedule of 2001/2002 and contains all trains operated by Deutsche Bahn. The European network is based on the winter schedule of 1996/1997

Table 4.11. Comparison of base RAPTOR, rRAPTOR, LD, MLC, and SPCS on other instances. A trailing "8" in the algorithm description refers to a parallel execution on eight cores.

					Los Angeles		New York		Germany		Europe	
Algorithm	Arr.	Tran.	Rng.	Rel.	# Comp. p. Stop	Time [ms]	# Comp. p. Stop	Time [ms]	# Comp. p. Stop	Time [ms]	# Comp. p. Stop	Time [ms]
RAPTOR	●	●	○	○	11.5	2.5	7.1	2.3	36.5	5.4	40.4	42.2
RAPTOR-8	●	●	○	○	11.6	1.4	7.2	1.5	37.0	2.4	40.9	16.1
TD	●	○	○	○	7.3	5.2	4.8	4.9	26.2	8.4	23.8	50.9
LD	●	●	○	○	13.5	14.8	9.6	13.3	55.5	26.9	68.9	207.3
MLC	●	●	○	○	16.6	27.9	13.5	22.0	82.4	61.9	152.4	462.7
rRAPTOR	●	●	●	○	46.1	12.7	41.1	16.6	68.6	10.8	54.5	58.9
rRAPTOR-8	●	●	●	○	56.2	9.3	51.2	7.8	167.4	11.8	135.6	50.5
SPCS	●	○	●	○	35.9	39.7	33.3	48.0	69.5	34.4	44.9	118.9
SPCS-8	●	○	●	○	52.7	18.8	54.4	17.7	185.5	25.5	134.8	92.5
McRAPTOR	●	●	○	●	36.6	22.9	43.2	29.4	81.3	28.3	90.3	189.0
MLC	●	●	○	●	73.4	69.0	115.4	82.3	180.7	114.5	329.0	839.7

and contains mostly long-distance trains. Figures for the sizes of all networks are summarized in Table 4.7 on Page 107. Note that while Los Angeles and New York are the biggest publicly available GTFS networks at the time of writing, they are both smaller than the London instance.

Moreover, for the metropolitan networks footpath data was not available to us. Hence, we generated footpaths on these instances with our heuristic from Section 4.3.5. It creates cliques of footpaths between stops that are close to the same intersection of the road network. For Germany and Europe we use minimum change times to model transfers within railway stations. Since no fare zone data is readily available for any of the networks, we did not run our multicriteria algorithms that include fare zones.

Table 4.11 shows the results for all relevant algorithms. The results are consistent with the previous experiments: RAPTOR outperforms both LD and MLC on every instance. It can compute all Pareto-optimal journeys between two random stops within 2.5 ms on Los Angeles, 2.3 ms on New York, and 5.4 ms on Germany. On Europe, RAPTOR takes 42.2 ms, which is due to the very high number of routes in this instance (44 751 compared to, e. g., 2 240 on London, cf. Table 4.7). Parallelizing RAPTOR shows moderate effect: Speedups are below a factor of 2.6 for eight cores on all instances.

McRAPTOR (with reliability) computes journeys in under 30 ms on all instances except Europe, where it takes 189 ms.

Running rRAPTOR results in query times below 17 ms for all instances except Europe, where it is 59 ms. Parallelizing rRAPTOR only pays off on the dense urban networks, but speedups are still limited. On the two railway networks parallel

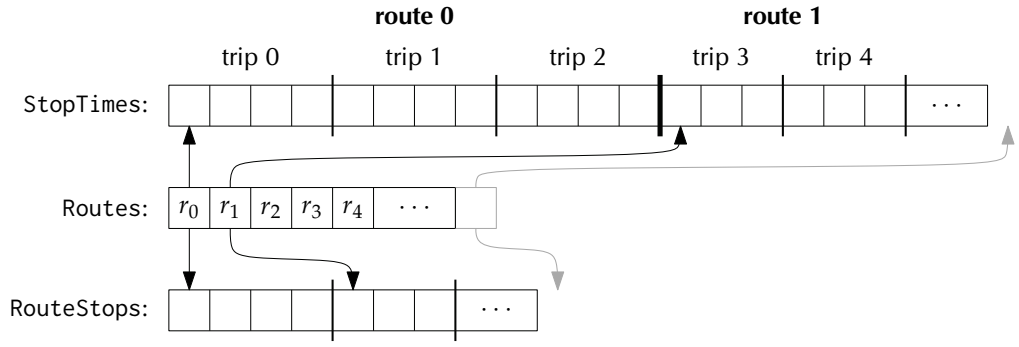


Figure 4.28. Illustration of the adjacency structure of routes.

rRAPTOR does not scale. Here, trips operate too infrequently, thus, too few of them can be distributed among the cores. Finally, we observe that rRAPTOR again outperforms SPCS by a factor of up to 3.2.

4.6.9. Implementation Details

In this section, we present details on the data structures we use for RAPTOR. For simplicity, we assume all routes, trips, and stops have sequential integral identifiers, each starting at 0.

Route Traversal. For the main loop of the algorithm, we need to traverse routes. For route r, we need its sequence of stops (in order), as well as the list of all trips (from earliest to latest) that operate on that route.

To accomplish this, we store an array `Routes` where the i-th entry holds information about route r_i. It stores the number of trips associated with r_i, as well as the number of stops in the route (which is the same for all its trips). It also stores pointers to two lists. The first pointer in `Routes`$[i]$ is to a list representing the sequence of *stops* along route r_i. Instead of representing each list of stops separately (one for each route), we group them into a single array `RouteStops`. Its first entries are the sequence of stops of route 0, then those for route 1, and so on. The pointer in `Routes`$[i]$ is to the first entry in `RouteStops` that refers to route i. See Figure 4.28. The second pointer in `Routes`$[i]$ points to a representation of the list of *trips* that operate on that route. Once again, instead of keeping separate lists for different routes, we keep a single array `StopTimes`. (See Figure 4.28.) This array is divided into *blocks*, and the i-th block contains all trips corresponding to route r_i. Within a block, trips are sorted by departure time (at the first stop). Each trip is just a sequence of *stop times*, represented by the corresponding arrival and departure times.

A route r_i can be processed by traversing the stops in `RouteStops` associated with r_i. To find the earliest trip departing from some stop p along the route after some time τ,

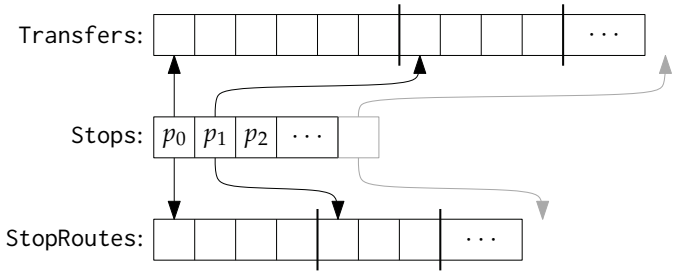


Figure 4.29. Illustration of the adjacency structure of stops.

we can quickly access the stop times of all trips at r_i at p in constant time per trip due to the way we sorted `StopTimes`. In particular, when processing r_i with trip t, the arrival time of the next stop is determined by the subsequent entry in `StopTimes`. Furthermore, to check for an earlier valid trip of r_i, we jump $|r_i|$ (the length of the route r_i, which is stored in `Routes`) entries to the left to retrieve the departure time of the next earlier trip. Note that these data structures are also suited for computing latest departure queries (cf. Section 4.2).

Timetable Compression. To enable timetable compression (cf. Section 4.6.5) in our data structure, we store the periodicity and size of the respective trip group with each stop time. (Recall that each stop time is part of a unique trip.) For each trip group, only the first (earliest) trip is represented in `StopTimes`. All subsequent trips (of that trip group) are stored implicitly and are, thus, discarded. Since every stop time knows about the periodicity and size of its trip group, the uncompressed stop times can be easily reconstructed on the fly.

Note that the data structure could be compressed even further: Instead of storing the periodicity and the size of a trip group at each stop time (which is redundant), we could store them only once in a separate array indexed by trip id. However for simplicity, we did not implement this approach.

Other Operations. We still need to support some operations outside the main loop of the algorithm. For those, we need an array `Stops`, which contains information about each individual stop. In particular, for each stop p_i, we must know the list of all routes that serve it in order to mark the appropriate routes between rounds. Moreover, we also need the list of all footpaths that can be taken out of p_i, together with their corresponding lengths.

As before, we aggregate these two sets of lists in two arrays. `StopRoutes` contains the lists of routes associated with each stop: First the routes associated with p_0, than those associated with p_1, and so on. Similarly, `Transfers` represents the allowed footpaths from p_0, followed by the allowed footpaths from p_1, and so on. (Each individual footpath from p_i is represented by its target stop p_j together with the

transfer time $\ell(p_i, p_j)$.) The i-th entry in `Stops` points to the first entries in `StopRoutes` and `Transfers` associated with stop p_i. See Figure 4.29.

Bags. Some of our algorithms, such as McRAPTOR, use bags to represent sets of nondominated labels. Each bag B_i is represented by a dynamic (unordered) array which contains its labels. Whenever we merge another bag B_j into B_i, we check domination between each pair of labels $L_i \in B_i$ and $L_j \in B_j$, adding labels to B_i accordingly. Dominated labels in B_i are marked and erased from the array at the end of the merge operation.

4.6.10. Conclusion

In this section, we have introduced RAPTOR, a new algorithm for fast multicriteria journey planning in public transit networks. Unlike previous algorithms, it neither operates on a graph nor requires a priority queue. Instead, it exploits the inherent structure of such networks by operating in rounds and processing each route of the network at most once per round. Moreover we extend it to range queries, and additional criteria (such as fare zones and reliability of transfers) can be added. Experiments on the transit network of London reveal that RAPTOR is more than an order of magnitude faster than previous approaches. RAPTOR can be easily parallelized, which accelerates queries even further. Finally, since RAPTOR does not rely on preprocessing, it can be directly used in dynamic scenarios, easily handling delays and trip cancellations.

Regarding future work, we are interested to handle networks of larger scale with RAPTOR, i. e., inhomogeneous networks which contain (many) local transit agencies, interconnected by long-distance travel, e. g., the network of a full country. Since RAPTOR's performance is determined by the number of scanned routes (per round), preprocessing may be necessary to achieve fast queries. In particular, we are interested to apply separator-based techniques, similarly to [SWW00, SWZ02], to RAPTOR.

Another open problem is whether uncertainty (beyond optimizing reliability) can be incorporated into RAPTOR. In particular, minimizing the expected arrival time, similarly to CSA [DPSW13], would be interesting.

Finally, further investigating the computation of full Pareto sets using strict domination (see Section 4.6.3) would be interesting. One challenge would be to identify significant alternative journeys from the larger Pareto set for presentation to the user. Alternatively, for dense metropolitan transit networks (where—at least in parts—trips operate very frequently), an entirely different form of presentation (instead of individual journeys) could be more useful to the user. For examples, for travelers that want to get from Karlsruhe's main railway station to KIT, one could suggest to take either the routes S1, S4, or Tram 2 (whatever arrives first), and then transfer at "Marktplatz" into Tram 4, Tram 5, or S2 (whatever arrives first), or take Tram 4

from the main station directly. Automatically computing such "guidebook routing" schemes from the full Pareto set would be an interesting problem.

Chapter 5

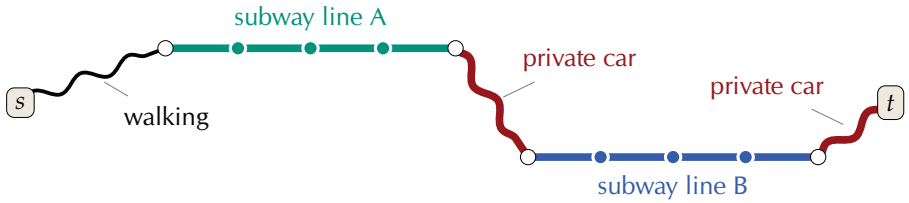
Multimodal Journey Planning

Multimodal route planning refers to the problem where one is interested in computing journeys that involve various different modal networks (such as public transit, car/taxi travel, walking, and rental bicycles) in a *single* and *integrated* query. Thereby, the main challenge is handling modal transfers. In particular, a naïve approach that "merges" the networks and then simply applies a shortest path algorithm that, e. g., solves the earliest arrival problem (as in Section 4.2), might result in very undesirable journeys. For example, it might produce a journey where the user is required to take a private car (or taxi) for a long segment between two train trips. Clearly, taking a private vehicle is impossible, since the user will not have it available at that point of the journey. Using a taxi instead might still be undesirable for some users.

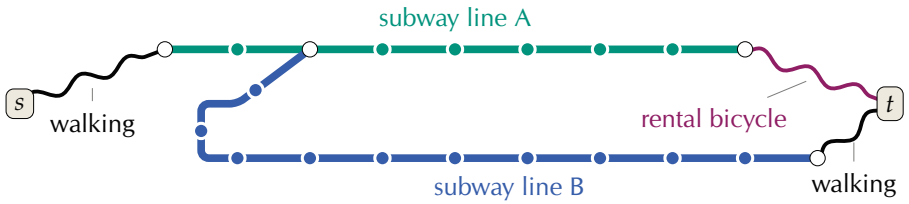
Figure 5.1 illustrates this problem in a metropolitan scenario including the following transportation modes: Public transit, cars, walking, and rental bicycles. While the journey that minimizes the earliest arrival time from s to t requires the user to take a car two times, once between the subway rides, and again at the end; two preferable alternatives (obviously, with somewhat higher arrival time at t) also exist: The first involves some walking in the end, and the second requires a rental bicycle on the last leg of the journey.

This chapter deals with multimodal route planning algorithms that take modal transfers into account, explicitly considering these issues.

Inputs. To be able to design an algorithm, we must first describe what transportation networks we consider and how they are modeled in a mathematical sense. While for transit networks Chapter 4 already defined both the input (Section 4.1) and several graph-based models (Section 4.3), we here define road networks (for car/taxi as well as pedestrians), flight networks, and rental bicycle networks. Having established each individual transportation network, we also explain how we combine them to obtain



(a) Earliest arrival journey.



(b) More desirable alternative journeys.

Figure 5.1. Journeys from s to t having different transportation mode sequences. The earliest arrival journey (top) requires a private car between the two subway trips and also as the last leg. However, this might not be desired by the user. The bottom two (slightly longer) alternative journeys may be more convenient. The first has a direct transfer to subway line B with walking (instead of a car) as last leg. The other uses subway line A longer and a rental bicycle as last leg.

an integrated multimodal network, for which we then design algorithms. Note that this integrated multimodal network is inherently time-dependent, whenever a public transit network is part of it.

Problems and Basic Algorithms. To obtain multimodal journeys between arbitrary locations (in any of the networks), one could now simply consider one of the problems from the previous chapter, such as the earliest arrival problem (cf. Section 4.2). In fact, all of the problems mentioned in Section 4.2 carry over to the multimodal scenario. However, they do not take *modal* transfers into account explicitly. We therefore recap the well-known *label-constrained shortest path problem* (LCSPP) [BJM00]. Roughly speaking, it defines an alphabet of transportation modes, and arcs in the multimodal network are labeled by symbols from the alphabet that correspond to their respective transportation mode. Then, the concatenation of the labels along *any* (shortest) path must obey constraints defined in terms of a formal language (an input). For the case of regular languages (cf. Section 3.3), Dijkstra's algorithm can be extended to solve the LCSPP [BJM00]. Finally, we consider a subset of regular languages that is sufficient for the scenario of multimodal route planning, called *mode sequence constraints*: They specify constraints on *sequences* of transportation modes rather than constraints on individual arcs.

User-Constrained Contraction Hierarchies. Having established the label-constrained shortest path problem (LCSPP) we present the User-Constrained Contraction Hierarchies (UCCH) algorithm. It is the first multimodal speedup technique that handles arbitrary mode sequence constraints as input to the query (rather than to the preprocessing) and requires significantly less preprocessing effort than previous techniques with similar query performance, such as Access-Node Routing [DPW09a].

To this extent, we revisit one particular technique, namely *vertex contraction*, that has proven successful in road networks in the form of Contraction Hierarchies, introduced by Geisberger et al. [GSSV12]. By ensuring that shortcuts never span multiple modes of transport, we extend Contraction Hierarchies in a sound manner. Moreover, we show how careful engineering further helps our scenario.

Our experimental study is based on an intercontinental instance composed of cars, railways and flights with over 50 million vertices, 125 million arcs, and 30 thousand stations. With only 557 MiB of auxiliary data, we achieve query times that are fast enough for interactive scenarios.

Multimodal RAPTOR. While label constraints can be used to restrict sequences of transportation modes, a crucial disadvantage of this approach is that the user has to specify (and therefore know) reasonable constraints *in advance*. Given these limitation, we revisit the problem of finding multicriteria multimodal journeys on a metropolitan scale. Instead of optimizing each mode of transportation independently [EL11], we argue that most users optimize three criteria: travel time, convenience, and costs. While this produces a large Pareto set, we propose using fuzzy logic [FA04, Zad88] to filter it in a principled way to a modest-sized set of representative journeys.

This postprocessing step is not only quick, but can also be user-dependent, incorporating personal preferences. Combining RAPTOR (Section 4.6.6) with UCCH (Section 5.3) into an algorithm called Multimodal Multicriteria RAPTOR (MCR) allows us to answer exact queries optimizing time and convenience in less than two seconds within a large metropolitan area (for a simpler scenario of walking, cycling, and public transit). Unfortunately, this is not enough for interactive applications and becomes much slower when additional criteria, such as costs, are incorporated. We therefore also propose heuristics (still multicriteria) that are significantly faster and closely match the top journeys in the Pareto set. A thorough experimental evaluation of all algorithms in terms of both solution quality and performance shows that our approach can be fast enough for interactive applications. Moreover, since it does not rely on heavy preprocessing, it can be used in fully dynamic scenarios.

Overview. Section 5.1 formally defines the inputs to each transportation network we are using in the following chapter and explains how they are combined to obtain an integrated network. Section 5.2 recaps the label-constrained shortest path problem and shows how Dijkstra's algorithm can be extended to it. In Section 5.3 we introduce

UCCH, our speedup technique for the label-constrained shortest path problem. Finally, Section 5.4 presents MCR, our multicriteria multimodal approach.

5.1. Inputs

This section precisely defines our input to the multimodal route planning problems considered in this chapter. We, therefore, introduce each modal (sub)network individually, before we explain how we combine them into a holistic multimodal network that then enables transfers *between* different modes of transportation. The types of networks considered in this chapter are street networks (for walking and cars), public transportation (cf. Chapter 4) that is used for local and long-range transit (such as busses, subways, and railways), flight networks (which can be regarded as a form of public transit, but with special properties), and rental bicycle schemes. We explain each network type in turn before turning toward combining them into a multimodal network.

5.1.1. Street Networks

The input to the transportation modes car and walking (sometimes we also refer to the walking mode by *foot*) is the *street network* which consists of a set of intersections and a set of street segments, in which each segments connects exactly two intersections. Street segments usually have different attributes, such as geographical length (in meters), functional street category (such as "highway" or "rural road"), flags indicating supported transportation modes (walking, car traffic, and bicycle traffic), and the (average) traffic speed for cars (in kilometers per hour).

From this data, we build a directed graph $G = (V, A)$, where each vertex $u \in V$ corresponds to an intersection, and an arc $(u, v) \in a$ connects two intersections, if the respective street segment supports the considered mode of transportation. (Note that, for example, pedestrian zones only support walking but no car traffic, while highways support car traffic but no walking.) Moreover, we define a cost function $\text{len}\colon A \to \mathbb{Z}_{\geq 0}$ on the arcs, which depicts the *travel time* (usually in seconds) along the respective arc. If the mode of transportation is car, we compute the arc's travel time from its associated average speed and geographical length. For pedestrians we assume a constant walking speed of 5 kph and for bicycles a constant riding speed of 12 kph.

Note that more complex street network models exist. In the context of route planning, *turn costs* (and turn restrictions) have been considered in [GV11, DGPW11] and, also, in Chapter 6. They can be incorporated by either blowing up the graph at each intersection that exhibits turn costs, or, by maintaining turn costs in a separate data structure and adapting the query algorithm. Even more realistic models consider *polyvalent turn costs*, which model dependencies between intersections [Sas11, Sch12]: Depending on the turn a driver takes at some intersection, the cost may change at the

next. An example for such restrictions is to forbid paths that leave and immediately re-enter highways at the same exit. Polyvalent turn restrictions can also be incorporated by either blowing up the graph or by adapting the query algorithm [Sas11]. In our model we use neither turn costs nor polyvalent turn restrictions, since they were not available from our input data.

5.1.2. Public Transit Networks

For the mode of *public transit*, the input (i. e., timetables) as well as various models with different levels of realism have been discussed in Section 4.3 extensively. If not stated otherwise, we use the *realistic time-dependent model* [PSWZ04b, PSWZ08] in this chapter. Note that it keeps a *stop vertex* for each stop of the timetable and, at each stop p, a *route vertex* for every route that serves the stop p. Arcs connect stop vertices p to their associated route vertices with constant cost depicting the minimum change time required for transferring at p. Moreover, subsequent route vertices (of each route in the timetable) are connected by arcs with *time-dependent* cost that map departure times of the day to travel times. See Section 4.3.3 for details.

5.1.3. Flight Networks

Regarding the *flight* mode of transportation, the input is a *flight schedule* which is very similar to timetables (which have been introduced in Section 4.1) for the mode of public transit. Following [Paj09, DPWZ09], a flight schedule $\mathfrak{F} = (\Pi, \mathcal{S}, \mathcal{T}, \mathcal{R})$ is defined in terms of a set $\mathcal{S}$ of *airports* (equivalent to the stops of a public transit timetable), a set $\mathcal{T}$ of *trips* (or *flights*), and a set $\mathcal{R}$ of *routes*. Just like in timetables, $\Pi \subset \mathbb{Z}_{\geq 0}$ depicts the *period of operation*. Any trip $r \in \mathcal{R}$ defines—in addition to the airports it serves—an associated *flight alliance*, denoted by alliance(r). Based on this input any of the graph-based models from Section 4.3, such as the realistic time-dependent model, could be used. However, realistic railway models tend to produce graphs of (unnecessarily) large size: In contrast to public transit networks, routes in flight networks are short (in fact, most of them have only a single hop). On the other hand, airports usually serve a great amount of different destinations, which would result in a very large number of route vertices per airport in the graph.

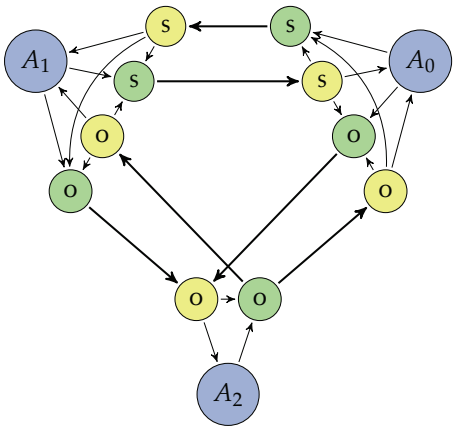


Figure 5.2. Realistic flight model graph for two flight alliances: Star Alliance (s) and Oneworld (o) [DPWZ09].

To overcome these issues, a tailored realistic time-dependent *flight model* is introduced in [DPWZ09], which we also use in this chapter. Inspired by the time-dependent railway model, it creates, for each airport, a single *airport vertex* $p \in V$.

Then, for every flight alliance served by p, it creates dedicated *arrival* and *departure vertices*. The airport vertex, and the arrival and departure vertices of each flight alliance, are interconnected by arcs with constant cost reflecting the different times required for airport procedures such as check-in, check-out, and transfers within and between flight alliances. To model travel, the model creates, for each route r and each two subsequent airports p_1, p_2 served by r, a time-dependent arc (u, v) between the departure and arrival vertices of flight alliance alliance(r) at p_1 and p_2. Equivalently to public transit networks, the cost of route arcs encodes the flight schedule of the respective route as a piecewise linear travel time function, see Section 4.3.3. The complexity of the resulting flight model graph is significantly lower than for the realistic time-dependent railway model: The number of vertices (and arcs) at every airport is linear in the number of flight alliances (rather than routes). See [DPWZ09] for details.

An illustration of the realistic flight model is shown in Figure 5.2. The network contains three airports and two flight alliances: Star Alliance (s) and Oneworld (o). Departure vertices are drawn in green while arrival vertices are yellow. Moreover, vertex labels depict their associated flight alliance.

5.1.4. Rental Bicycle Schemes

Another mode of transportation we consider in this chapter is *rental bicycles*. (Note that *private* bicycles are just a form of street travel, which we already discussed.) The input to this mode of transportation is the street network and a set $\mathcal{S}$ of *rental bicycle stations* that are spread over the street network at different locations. A common scheme (which is for example in use by Transport for London [Tra00]) is the following: A user may take a bicycle from any of the bicycle stations $\mathcal{S}$ and ride it to any other station. Usually there is a small fee per pick-up, as well as an additional fee depending on the riding duration.

Our graph models the street network equivalently to the car and walking modes of transportation: Vertices depict intersections and arcs represent street segments that are open for bicycles. Arc costs depict travel times which are obtained from the geographical length of the street segment and an assumed average riding speed of 12 kph. To model bicycle stations, we add an extra vertex per station $p \in \mathcal{S}$ and connect it to the geographically closest vertex of the street network (with cost computed as before). Note that the extent of the bicycle network is limited: Since *any* bicycle journey begins and ends at a bicycle station, the graph only needs to cover (the area that includes) all shortest paths between pairs of bicycle stations.

Also note that other rental bicycle schemes exist that allow the user to drop their bicycle at any "reasonable" location (such as major street intersections) and, thus, do not employ special stations. However, in this chapter we focus on rental schemes where the bicycle must be always picked up and dropped off at one of the stations.

5.1.5. Combining the Networks

Having introduced the models we use for each individual mode of transportation, we now describe how we combine them into a multimodal graph. More precisely, we create a *multimodal graph* $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{A})$ (we use bold letters to refer to multimodal graphs) by *merging* the individual graphs of the subnetworks. Let $G_1 = (V_1, A_1), \ldots, G_k = (V_k, A_k)$ be graphs of k subnetworks, then we simply set $\boldsymbol{V} = V_1 \cup \cdots \cup V_k$ and $\boldsymbol{A} = A_1 \cup \cdots \cup A_k$. However, this results in k disconnected subgraphs in $\boldsymbol{G}$. To enable transfers between different modes of transportation, we add *link arcs*: We link every stop vertex of the public transit network and every station vertex of the bicycle network to its geographically closest vertex in both the walking and car networks, unless the distance exceeds a certain threshold (we use 500 meters). Similarly, we link airport vertices from the flight network to the walking and car networks. In addition, we also link each airport to its geographically closest stop vertex in the railway network, unless the link distance exceeds 5 000 m. Arc costs of the inserted link arcs are computed from the geographical length of the arc and an assumed average pedestrian walking speed of 5 kph. To compute accurate distances, we compute the geodesic length on the GRS80-ellipsoid [Mor92], which is also used by the Global Positioning System (GPS). (Note that all our vertices have associated longitude and latitude coordinates.)

Some of the algorithms in Section 5.4 do not use an *integrated* graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{A})$. Instead, they consider each subnetwork individually. To still enable modal transfers, we *identify* linked vertices in different networks but keep them separate. While this approach does not incur any cost (i. e., travel time) for modal transfers, the introduced error is small in practice (recall that link arcs in $\boldsymbol{G}$ connect geographically closest vertices).

5.2. Problems and Basic Algorithms

This section formally introduces the multimodal route planning problems we consider in this chapter. The earliest arrival and multicriteria problems (Sections 5.2.1 and 5.2.2) are essentially equal to their counterparts in the context of public transit networks (see Section 4.2). However, they do not constrain modal transfers. Therefore, we recap the label-constrained shortest path problem [BJM00] in Section 5.2.3 and present an algorithm that computes label-constrained shortest paths for constraints that are given by regular languages.

5.2.1. Earliest Arrival Problem

Similarly to public transit route planning (Chapter 4), the simplest problem we are considering in the context of multimodal route planning is the *earliest arrival problem*. Given a multimodal network (for example as a multimodal graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{A})$) and

source and target *locations* $s, t \in \boldsymbol{V}$, as well as a departure time τ (recall that $\boldsymbol{G}$ is time-dependent as soon as it contains public transit as a subnetwork), it asks for a journey from s to t that departs s no earlier than τ and arrives at t as *early as possible*. Again, an algorithm that solves the earliest arrival problem is also called *earliest arrival query*.

Equivalently to public transit route planning, the solution consists of (at most) one (optimal) journey, namely the one which arrives at t earliest. If more than one optimal journey exists, we break ties arbitrarily. Note that nothing is said (so far) about the involved modes of transportation of the journey. In particular, it may contain arbitrary transfers between any modes of transportation, at any point of the journey. Clearly, this is a big disadvantage (from a practical point of view) concerning earliest arrival queries in multimodal route planning. Also recall Figure 5.1a.

Algorithms. The earliest arrival problem in multimodal networks can be solved by a time-dependent variant of Dijkstra's algorithm (equivalently to the scenario of public transit route planning) [Dij59, CH66]. It works on the combined multimodal graph $\boldsymbol{G}$ and maintains for each vertex $u \in \boldsymbol{V}$ a label depicting the (tentative) earliest arrival time at u. It uses a priority queue Q to scan vertices in increasing order of arrival time. Each time a vertex $u \in \boldsymbol{V}$ is scanned, it relaxes its incident arcs $a = (u, v) \in \boldsymbol{A}$, thereby, minimizing the arrival time at v (and updating Q, accordingly). Section 4.4.1 contains more details and a running time analysis of Dijkstra's algorithm.

5.2.2. Multicriteria Problem

The multicriteria problem aims to remedy a key issue of the earliest arrival problem, namely, that it only outputs a single journey. Therefore, it considers (besides arrival time) further optimization criteria in order to compute a Pareto set of solutions that trade off these criteria. More formally, the *multicriteria problem* takes a multimodal network, source and target locations $s, t \in \boldsymbol{G}$, and a departure time τ as input. It asks for a *Pareto set* $\mathcal{J}$ of journeys that depart at s no earlier than τ and arrive at t. For each two journeys $J_1, J_2 \in \mathcal{J}$ it must hold, that neither J_1 dominates J_2, nor J_2 dominates J_1. Thereby, a journey J_1 *dominates* a journey J_2, if J_1 is better (or equal) in *all* criteria than J_2. Also note that the Pareto set $\mathcal{J}$ must be *maximal*: For a journey J to not be included in $\mathcal{J}$, there must be a witness journey $J' \in \mathcal{J}$ that dominates J. We also call an algorithm that solves the multimodal multicriteria problem *multimodal multicriteria query*. We consider the multicriteria problem in Section 5.4.

Algorithms. To solve the multicriteria problem, several algorithms have been discussed in the context of public transit route planning in Section 4.4.2. They can also be used in our scenario by applying them to the multimodal graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{A})$. The first algorithm is the *label correcting* algorithm (LC) [Dea99], which maintains a Pareto set of labels at each vertex $u \in \boldsymbol{V}$. It then scans, beginning at s, vertices in some

consistent order. Each time a vertex $u \in \boldsymbol{V}$ is scanned by the algorithm, it looks at each arc $(u, v) \in A$ and extends *all* labels from u over a (adding the cost of a to every label). The resulting set of labels is then *merged* into the maintained Pareto set at v, eliminating dominated labels on the fly. If this resulted in the insertion of a new label at v, the vertex must be (re)considered for scanning by the algorithm.

The multi-label-correcting algorithm (MLC) [PSWZ08, DMS08] works similarly, but instead of considering the whole Pareto set of labels of a vertex u (at the time of its scan), it looks at labels individually: It maintains a priority queue Q of labels (and associated vertices) and extracts, in each iteration, a minimum *label* (according to some consistent ordering). This improves LC by not scanning the same label at the same vertex more than once.

Finally, for the special case of bi-criteria optimization, where one of the criteria is discrete and only assumes a small set of different values, the layered Dijkstra algorithm [BJ04] may be more efficient. If K is a bound on the maximum value of the discrete criterion, it copies the graph $\boldsymbol{G}$ into K layers $\boldsymbol{G}_1, \ldots, \boldsymbol{G}_k$. In each layer k, arcs that exhibit cost on the discrete criterion are rewired such that they point one layer upward. Then, computing an *earliest arrival query* from $s_0 \in \boldsymbol{V}_0$ to the target vertex t_K for all layers $k \leq K$ is results in a Pareto set of labels at the vertices t_K ($k \leq K$).

Several optimizations, such as local and target pruning, hopping reduction, and label forwarding, exist for the multicriteria algorithms. We refer to Section 4.4.2 for more details.

5.2.3. Label-Constrained Shortest Path Problem

While the problems from Section 5.2.1 and 5.2.2 can both be used to obtain journeys in a mutlimodal network, they (and in particular the earliest arrival problem) have one major disadvantage: Neither modal transfers nor the sequence of transportation modes is considered during query. As a result, the obtained journeys may require the user to take arbitrary means of transportation in an arbitrary order. (Actually, in Section 5.4 we use the multicriteria problem to obtain diverse solution sets that reasonable trade off the usage of different means of transportation.) While for some journeys this may just be inconvenient, for others it may even be infeasible for the user. Reconsidering the example in Figure 5.1 on Page 120, the depicted earliest arrival journey may require the user to drive a private car at two undesirable locations: Between the subway rides and on the last leg of the journey. If a car is not available at these points of the journey, this solution is of no value to the user. Therefore, we are interested in computing journeys, where modal transfers between transportation modes can be *restricted* (ideally by the user). For example, Figure 5.1b shows two earliest arrival journeys where the order of the means of transportation is restricted to (a), walking, public transit, and walking, or (b), walking, public transit, and taking a rental bicycle.

An elegant approach to model such restrictions is the *label-constrained shortest path*

problem (LCSPP). It has been first introduced in [BJM00] and has been applied in the context of multimodal route planning in [BJM00, BBJ+02, BBH+09, Paj09, DPW09a, KLPC11, KLC12, RT10]. Its general formal definition is as follows. We are given as input an alphabet Σ, a Σ-labeled graph $G = (V, A)$, that is, each arc $a \in A$ is associated with a label from Σ denoted by $\sigma(a) \in \Sigma$, as well as, source and target vertices $s, t \in V$. Moreover, we are also given a language $L \subset \Sigma^*$ as input. The problem now asks for a shortest path in G from s to t, however, with the following restriction: Any (shortest) path must obey the constraints given by the language L. More formally, let P be a path in G, then the word w formed by concatenating the (associated) labels along the arcs of P must be contained in L. We also call such a path L-constrained.

The general formulation of the label-constrained shortest path problem states no restriction on the language L given as input. The computational complexity of LCSPP for different types of languages has been studied in [BJM00]. In particular, for *regular* languages L, the problem is solvable in (deterministic) polynomial time. Note that in this case we may—instead of a language L—give the corresponding (nondeterministic) finite automaton $\mathcal{A}_L$ as input. In fact, we use L and $\mathcal{A}_L$ interchangeably whenever we consider regular languages as input.

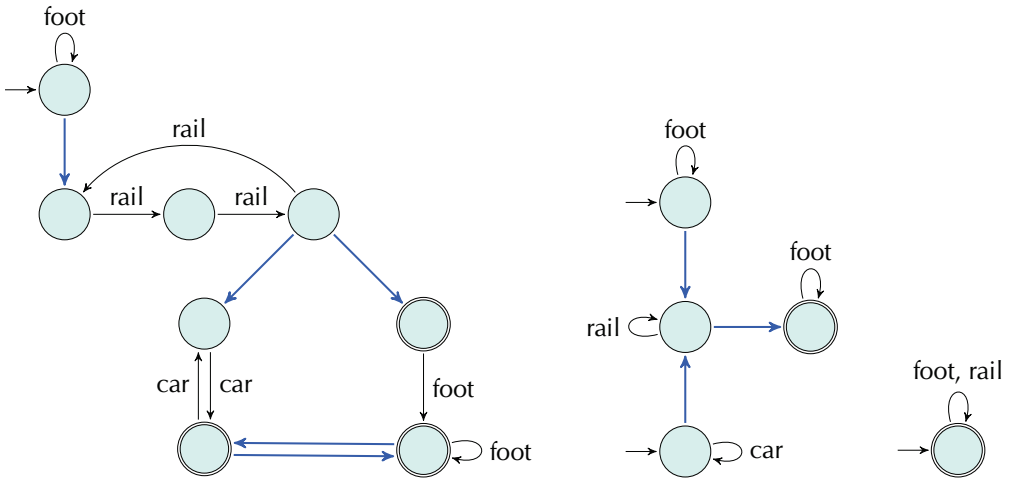
We apply LCSPP to the scenario of multimodal route planning as follows. Let $\boldsymbol{G}$ be the multimodal graph according to Section 5.1. We now define Σ to contain a designated symbol for each considered mode of transportation, for example,

$$\Sigma = \{\texttt{foot}, \texttt{car}, \texttt{rail}, \texttt{flight}, \texttt{link}\}. \tag{5.1}$$

Moreover, we label every arc $a \in \boldsymbol{A}$ with the symbol that reflects the mode of transportation a belongs to. Then a regular language over Σ restricts modal transfers in $\boldsymbol{G}$. Note that since $\boldsymbol{G}$ is time-dependent, an additional input to the problem—besides source and target vertices $s, t \in \boldsymbol{V}$—is the departure time τ. We, hence, ask for a L-constrained path from s to t that leaves s no earlier than τ and arrives at t as early as possible.

Mode Sequence Constraints. A subset of regular languages which is reasonable in the context of multimodal route planning are languages that model *mode sequence constraints* (LCSPP-MS). More formally, a regular language L models mode sequence constraints if for any word $w = \sigma_1 \cdots \sigma_k \in L$ and any symbol $\sigma_i \in w$ that is other than `link` the word $\sigma_1 \cdots \sigma_i^j \cdots \sigma_k$ is also contained in L for any $j \geq 0$. Moreover, none of the words may contain two (or more) consecutive symbols $\sigma = \texttt{link}$ (recall that we use `link` for representing modal transfers). In other words, L restricts *sequences* of transport modes, but it does not restrict travel *within* a transport mode.

Note that an even more restricted form of regular languages are *Kleene languages*. Given an alphabet Σ, they are of the form $(\Xi)^*$, where $\Xi \subseteq \Sigma$ is a subalphabet. They are used to (globally) exclude certain means of transportation. However, they are not



(a) Arbitrary regular language. (b) Mode sequence constraints. (c) Kleene language.

Figure 5.3. Exemplary nondeterministic finite automata representing three types of regular language constraints. General regular languages (left) allow arbitrary restrictions of travel, even within a transport mode. Mode sequence constraints (middle) restrict sequences of transport modes, while Kleene languages (right) include (or exclude) certain modes of transportation globally. Blue arcs are labeled `link`.

powerful enough to define constraints on the *order* of transportation modes along journeys. Kleene languages have been studied in the context of multimodal route planning in [RT10].

Figure 5.3 shows exemplary nondeterministic finite automata that correspond to the three types of regular languages: A general regular language automaton (Figure 5.3a), a regular language automaton modeling mode sequence constraints (Figure 5.3b), and a Kleene language automaton (Figure 5.3c).

Label-Constrained Dijkstra. Besides showing that the label-constrained shortest path problem on regular languages is solvable in polynomial time, in [BJM00] an algorithm that actually solves it is presented as well. Furthermore, it has been extended to time-dependent graphs in [Paj09] by combining it with time-dependent Dijkstra [CH66]. To solve the LCSPP, the algorithm works on the *product graph* of the input network $\boldsymbol{G}$ and the transition graph of the automaton $\mathcal{A}_L$ representing the input language L. The following definition captures the notion of product graphs.

Definition 2 (Product Graph)**.** *Given a Σ-labeled (multimodal) graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{A})$ and a nondeterministic finite automaton $\mathcal{A} = (S, \Sigma, \delta, I, F)$, the* product graph $G^\times = (V^\times, A^\times)$ *is defined as follows.*

- *The vertex set $V^\times$ is defined as $V^\times = \{(u, q) \mid u \in \boldsymbol{V}$ and $q \in S\}$, and*

- *the arc set $A^\times$ exactly consists of all arcs $a^\times = \big((u_1, q_1), (u_2, q_2)\big)$ for which $a = (u_1, u_2) \in \boldsymbol{G}$, and there exists a symbol $\sigma \in \Sigma$, such that $q_2 \in \delta(q_1, \sigma)$. Thereby, the cost of $a^\times$ is set to the cost of a, and the label $\sigma(a^\times)$ of $a^\times$ is set to σ.*

Now, given source and target vertices $s, t \in \boldsymbol{V}$ of the original graph as well as a departure time τ, the algorithm runs a *multi-source multi-target earliest arrival query* on the product graph $G^\times$. More precisely it runs time-dependent Dijkstra's algorithm (cf. Figure 4.11 in Section 4.4.1) on $G^\times$ with the following modifications. For *every* initial state $q \in I$ of the automaton $\mathcal{A}_L$ it initializes the respective arrival time label of the vertex (s, q) to τ. It then proceeds as usual, extracting product vertices (u, q) from the priority queue Q in increasing order of arrival times. Then, for any vertex $u \in \boldsymbol{V}$ the earliest arrival journey that obeys the regular language constraints is determined by the minimum label among those product vertices (u, q), for which $q \in F$ is a final state of the automaton $\mathcal{A}_L$. (Note that the algorithm actually computes earliest arrival journeys for each final state of the automaton at every vertex $u \in \boldsymbol{V}$.) We call this algorithm *label-constrained time-dependent Dijkstra* (LCSPP-TD). If one is only interested in the earliest arrival time for the vertex t, the algorithm may stop as soon as the *first* product vertex (t, q) for which $q \in F$ is a final state of $\mathcal{A}_L$ has been extracted from Q.

In practice, explicitly computing the product graph $G^\times$ is wasteful. Instead, the algorithm uses $\boldsymbol{G}$ and $\mathcal{A}_L$ to compute $G^\times$ on the fly. It, therefore, maintains at each vertex $u \in \boldsymbol{V}$ an array of $|S|$ labels (depicting earliest arrival times for each state of $\mathcal{A}_L$), with all entries initialized to infinity. The priority queue still keeps vertex-state tuples. Whenever the algorithm extracts such a tuple (u, q) from Q, it scans all arcs $a = (u, v) \in \boldsymbol{A}$ and, for each such arc, it takes its associated symbol $\sigma \in \Sigma$ and enumerates all (via σ) reachable states in $\mathcal{A}_L$. More precisely, for every state $q' \in \delta(q, \sigma)$ it computes the arrival time of the path to v (via a) and minimizes the label associated with q' in v's label array. If the algorithm improved the label, it updates the respective entry for (v, q') in Q, accordingly. Figure 5.4 illustrates the algorithm in pseudocode.

The running time of LCSPP-TD is basically that of Dijkstra's algorithm (see Section 4.4.1), however, determined by the size of the (implicit) product graph $G^\times$. Since $V^\times$ contains a vertex for every vertex-state pair of $\boldsymbol{V}$ and S, it has size $|\boldsymbol{V}||S|$. Moreover, the number of arcs in $A^\times$ is bounded by $|\boldsymbol{V}||A||\delta|$, since for every vertex $(u, q) \in V^\times$ there is an (outgoing) arc for every arc $a \in \boldsymbol{A}$ and every symbol $\sigma \in \Sigma$ for which there exists a transition from q in δ. Using binary heaps as priority queue (equivalently to Section 4.4.1), we thus obtain a total running time for LCSPP-TD of $\mathcal{O}\big(|\boldsymbol{V}||A||\delta| + |\boldsymbol{V}||S| \log(|\boldsymbol{V}||S|)\big)$.

Speedup Techniques for LCSPP. Unfortunately, LCSPP-TD is too slow to be practical for real-time queries (such as for interactive server scenarios). Therefore, several speedup techniques exist that accelerate LCSPP(-TD) by precomputing auxiliary data

```
// Input: Graph G = (V, A), source vertex s, target vertex t, departure time τ,
          nondeterministic finite automaton A = (S, Σ, δ, I, F)
// Side Effects: Earliest arrival times τ(u, q) at all vertices u ∈ V and states q ∈ S, if t = ⊥
                 or at t, otherwise
// Initialization of the algorithm
1  Q ← new PQueue()                                          // Create empty priority queue
2  τ(·, ·) ← ∞                                               // Initialize arrival time labels for each vertex and state
3  forall the initial states q ∈ I do                        // Initialize arrival time for each initial state
4      τ(s, q) ← τ
5      Q.Insert((s, q), τ)
   // Main loop
6  while not Q.Empty() do
7      (u, q) ← Q.ExtractMin()                               // Scan next vertex-state tuple
8      if u = t and q ∈ F then                               // Stopping criterion
9          stop;
10     forall the outgoing arcs a = (u, v) ∈ A do            // Scan outgoing arcs
11         forall the states q' ∈ δ(q, σ(a)) do              // Enumerate transitions
12             τ_tent(v, q') ← τ(u, q) + f_a(τ(u, q))        // Compute tentative arrival time at (v, q')
13             if τ_tent(v, q') < τ(v, q') then              // Improve arrival time at (v, q')?
14                 τ(v, q') ← τ_tent(v, q')                  // Update label of v and state q'
15                 if not Q.Contains((v, q')) then           // Update priority queue
16                     Q.Insert((v, q'), τ_tent(v, q'))
17                 else
18                     Q.DecreaseKey((v, q'), τ_tent(v, q'))
```

Figure 5.4. Label-Constrained Time-Dependent Dijkstra (LCSPP-TD).

in a preprocessing phase. See also Section 2.3.2. Because of the relevance to this section, we briefly recap the most important.

In [BBJ+02] a first experimental study for linear regular languages is conducted. In [Paj09] the adaption of basic ingredients (of road network speedup techniques), such as bidirectional search [Dan62], goal-direction [GH05, Lau09, HKMS09], and shortcuts [SWW00, SS05, GSSV12] to time-dependent label-constrained shortest paths is discussed. It is observed that besides the challenges imposed by time-dependenc, such as the unknown arrival time at t a priori, also the regular language constraints L are not known beforehand. This makes it nontrivial to adapt preprocessing such that it handles arbitrary constraints at query times correctly. Basic speedup techniques such as bidirectional search [Dan62], A* [HNR68], and the Sedgewick-

Vitter Heuristic [SV86] have been tested in the context of multimodal route planning in [Hol08, BBH+09]. In [KLPC11] an algorithm called SDALT is presented that is based on ALT [GH05] but uses state-dependent lower bounds. In [KLC12] SDALT is extended to a label-correcting approach that can handle incorrect "lower bounds" (which improves running times). Correctness is ensured by scanning vertices multiple times.

Of particular interest to this chapter is Access-Node Routing (ANR), which has been introduced in [DPW09a]. It is inspired by Transit Node Routing in road networks [BFM+07] and handles *hierarchical* mode sequence constraints, where the road network (i. e., walking and car) may only be used at the beginning or end of a journey (see the automaton in Figure 5.7c for an example). During preprocessing, the algorithm computes for every vertex $u \in \boldsymbol{V}$ of the road network a (preferably minimal) set of relevant *access nodes* together with their distances from/to u. The access nodes are defined such that the following condition holds. For any shortest s–t-path (at any departure time) that does not solely use the road network, the *first* vertex that is *not* part of the road network has to be an access node of s. Analogously the *last* vertex not part of the road network must be an access node of t. Then, during query the algorithm skips the road network by running a multi-source multi-target LCSPP-TD query between the access nodes of s and t, returning the globally minimal path. To determine whether the query is *local*, i. e., solely uses the road network, a separate algorithm (in [DPW09a] CHASE [BDS+10] is used) must (always) be run.

While Access-Node Routing achieves query times in the order of milliseconds on continental networks, it has high preprocessing time (several hours) and space consumption (several Gigabytes). Moreover, preprocessing time strongly depends on the size of the public transit and flight subnetworks of $\boldsymbol{G}$. Unfortunately, for large networks, ANRs preprocessing is only feasible if the public transit and flight subnetworks are rather sparse [DPW09a].

Besides these disadvantages, the fastest preprocessing-based acceleration techniques, such as ANR and SDALT, fix the regular language L at preprocessing. Therefore, the (costly) preprocessing phase must be executed for *every* language L that should be supported during query time. In particular, label constraints cannot be specified arbitrarily by the user as an input to the query.

5.3. User-Constrained Contraction Hierarchies

In this section we introduce User-Constrained Contraction Hierarchies (UCCH), a speedup technique which solves the label-constrained shortest path problem for mode *sequence* constraints (see Section 5.2.3). To this extent, we augment Contraction Hierarchies [GSSV12] to the multimodal scenario. Its key idea is to not contracting transfer vertices. Therefore, we ensure that shortcuts never span multiple modes of transports. This enables the specification of mode sequence constraints as an

input of the *query* (i. e., by the *user*), a feature unavailable from previous LCSPP speedup techniques. Our experimental evaluation justifies our approach by indicating that—besides enabling mode sequence constraints at query time—UCCH has faster preprocessing times and significantly less space overhead than previous techniques with comparable query performance (such as Access-Node Routing [DPW09a]).

Overview. The section is organized as follows. First, we briefly recap Contraction Hierarchies for unimodal networks (such as road networks) in Section 5.3.1, which is the basis of our algorithm. Next in Section 5.3.2 we discuss why a naïve adaption of Contraction Hierarchies to a multimodal network has somewhat complicated preprocessing and query algorithms and does not allow arbitrary sequence constraints as a query input. In Section 5.3.3, we introduce User-Constrained Contraction Hierarchies (UCCH), our new algorithm which enables flexible mode sequence constraints at query time. Section 5.3.4 discusses several improvements to the algorithm that accelerate its performance, while Section 5.3.5 presents a detailed experimental study of our algorithm. Finally, we conclude in Section 5.3.6.

References. This section is based on [DPW12c], which appeared at the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12) and [DPW12d] which is (at the time of writing) under review at the ACM Journal of Experimental Algorithmics. It is joint work with Julian Dibbelt and Dorothea Wagner.

Moreover we thank Daniel Delling for interesting discussions on multimodal route planning and Geisberger et al. for providing us their CH code from [GSSV12].

5.3.1. Contraction Hierarchies on Unimodal Networks

Our algorithm is based on Contraction Hierarchies (CH), a preprocessing-based speedup technique developed for road networks. It has been introduced by Geisberger et al. in [GSSD08,GSSV12]. Before we extend it to the multimodal scenario, we briefly recap both preprocessing and query in the following.

Given a weighted (with constant weights), directed graph $G = (V, A)$, preprocessing works by heuristically ordering the vertices of the graph by an *importance* value (a linear combination of arc expansion, number of contracted neighbors, among others). Then, all vertices are *contracted* in order of ascending importance. To contract a vertex $u \in V$, it is removed from G, and shortcuts are added between its neighbors to preserve distances between the remaining nodes. The index at which u has been removed is denoted by $\text{rank}(u)$. To determine if a shortcut (u, w) is added, a local search from u is run (without looking at v), until w is scanned. If $\text{len}(u, w) \leq \text{len}(u, v) + \text{len}(v, w)$, the shortcut (u, w) is *not* added.

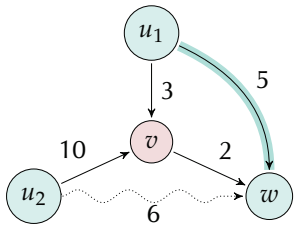


Figure 5.5. Illustrating vertex contraction.

The corresponding shorter path is called a *witness*. Figure 5.5 illustrates contraction of a vertex v. The highlighted arc (u_1, w) is added with length 5. The arc (u_2, w) is not needed since there exists an u_2–w-path with length 6, which is smaller than 12.

The CH query algorithm is a bidirectional Dijkstra search operating on G, augmented by the shortcuts computed during preprocessing. Each direction (forward or backward) searches "upward" in the hierarchy: The forward search only visits arcs (u, v) where $\text{rank}(u) \leq \text{rank}(v)$, and the backward search only visits arcs where $\text{rank}(u) \geq \text{rank}(v)$. Vertices where both searches meet represent candidate shortest paths with combined length μ. The algorithm minimizes μ, and each search can stop as soon as its minimum key of its priority queue exceeds μ.

Furthermore, we make use of the *stall-on-demand* technique [GSSV12]: When a vertex u is scanned in either search, we check for all its incident arcs $a = (u, v)$ of the *opposite* direction if $\text{dist}(v) + \text{len}(a) < \text{dist}(u)$ holds ($\text{dist}(v)$ denotes the tentative distance at v). If this is the case, the algorithm prunes the search at u, i. e., it does not scan any of its incident arcs. Also see [GSSV12] for details.

Partial Hierarchy. If the preprocessing is stopped prematurely, i. e., before all vertices are contracted, we obtain a *partial hierarchy* (pCH). For vertices which have not been contracted, we simply set their rank to infinity, i. e., $\text{rank}(u) = \infty$. Then, the same query algorithm as for Contraction Hierarchies is applicable and yields correct results. We call the induced subgraph of all uncontracted vertices *core* and the remaining (contracted) subgraph *component*. Note that both core and component can contain shortcuts not present in the original graph. Moreover, by definition of vertex contraction, the core is an *overlay graph*: For any pair of vertices u, v of the core, their distance $\text{dist}(u, v)$ in the core equals their distance in the original graph. (Recall that during contraction, we always ensure that distances are preserved in the remaining uncontracted graph.)

Performance. Both preprocessing and query performance of CH depend on the number of shortcuts added. It works well if the network has a pronounced hierarchy, i. e., if long shortest paths are covered by a sparse set of vertices [AFGW10, ADF+11] or if the network has small separators [BCRW13]. Note that if computing a complete hierarchy produces too many shortcuts, one can always stop preprocessing early and compute a partial hierarchy in practice. A possible stopping criterion is the *average vertex degree* in the core that is approached during the contraction process.

5.3.2. Contraction Hierarchies for Multimodal Networks

We now show how Contraction Hierarchies (CH) can be used to compute shortest path with restrictions on sequences of transport modes. We first argue that applying CH on the combined multimodal graph $\boldsymbol{G}$ without careful consideration either

yields incorrect results to LCSPP-MS or predetermines the automaton $\mathcal{A}$ during preprocessing. We then introduce User-Constrained Contraction Hierarchies (UCCH): A practical adaption of Contraction Hierarchies to LCSPP-MS that enables arbitrary modal sequence constraints as query input. Further improvements that help accelerating both preprocessing and queries are presented in Section 5.3.4.

Now, let $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{A})$ be a multimodal network as defined in Section 5.1. Recall that $\boldsymbol{G}$ is a combination of time-independent and time-dependent networks (for example, of road and public transit), hence, it contains arcs having both constant weights and travel time functions associated with them. Applying Contraction Hierarchies to $\boldsymbol{G}$, therefore, already requires some engineering effort: Shortcuts may represent paths containing arcs of different type. In order to compute the shortcuts' travel time functions, these arcs have to be *linked*, resulting in *inhomogeneous* functions that slow down both preprocessing and query performance. More significantly, when a path $P = (a_1, \dots, a_k)$ is composed into a single shortcut arc a', its labels need to be concatenated into a super label $L(a') = L(a_1) \cdots L(a_k)$. In particular, if there are subsequent arcs a_i, a_j in P where $L(a_i) \neq L(a_j)$, the shortcut induces a modal transfer. Running a query where this particular mode change is prohibited potentially yields incorrect results: The shortcut must not be used but the label constrained path (i. e. the one without this transfer) may have been discarded during preprocessing by the witness search (see Section 5.3.1). Note that the partial time-dependent nature of $\boldsymbol{G}$ further complicates things. A shortcut $a' = (u, v)$ needs to represent the travel time profile from u to v, that is, the underlying path P depends on the time of day. As a consequence, the super label of a' is time-dependent as well.

If the automaton $\mathcal{A}$ is known during preprocessing, we can modify Contraction Hierarchies preprocessing to yield correct query results with respect to $\mathcal{A}$. During contraction of vertex $v \in \boldsymbol{G}$ when the algorithm considers to add a shortcut $a' = (u, w)$, it looks at its super label $L(a') = (L_1, \dots, L_k)$. To determine if a' has to be inserted, it runs *multiple* witness searches as follows: For each state $q \in \mathcal{A}$ for which q represents $L(v)$, it runs a multimodal profile search from u (ignoring v). The algorithm runs it with q as initial state and all states $q' \in \mathcal{A}$ as final state who are reachable from q in $\mathcal{A}$ by applying $L(a')$. Only if for all these profile searches $\text{dist}(w) \leq \text{len}(a')$ holds, the shortcut a' is not required. (For every relevant transition sequence of the automaton, there is a shorter path in the graph.) Note that shortcuts $a' = (u, w)$ may be required even if an arc from u to w already existed before contraction. This results in parallel arcs for different subsequences of the constraint automaton.

This approach which we call *State-Dependent CH* (SDCH) has some disadvantages. First, witness search is slow and less effective than in the unimodal scenario, resulting in many more shortcuts. This hurts both preprocessing and query performance. Adding to it the more complicated data structures required for inhomogeneous travel time functions and arbitrary label sequences, SDCH combines challenges of both Flexible CH [GKS10] and Timetable CH [Gei10]. As a result we expect a significant

slowdown over unimodal Contraction Hierarchies on road networks. Most notably, however, SDCH predetermines the automaton $\mathcal{A}$ during preprocessing, which we want to avoid.

5.3.3. UCCH: Contraction for User-Constrained Route Planning

We now introduce User-Constrained Contraction Hierarchies (UCCH). Unlike SDCH, it can handle arbitrary sequence constraint automata during query and has an easier witness search. We first turn toward preprocessing before we go into detail about the query algorithm.

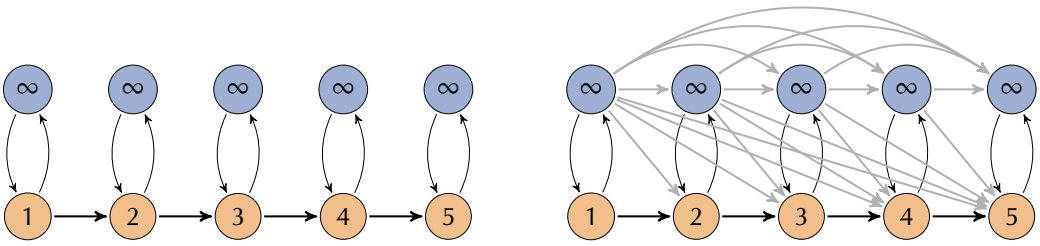
Preprocessing. The main reason behind the disadvantages discussed in Section 5.3.1 is the computation of shortcuts that span over boundaries of different modal networks.

Instead, let Σ be the alphabet of labels of a multimodal graph $\boldsymbol{G}$. We now process each subnetwork independently. We compute—in no particular order—a *partial* Contraction Hierarchy restricted to the subgraph $G_L = (V_L, A_L)$ (for every $L \in \Sigma$). Here, G_L is exactly the original graph of the particular transportation mode (before merging). We keep the contraction order with the exception of *transfer vertices*: Vertices which are incident to at least one arc labeled `link` in $\boldsymbol{G}$. We fix the rank of all such vertices u to infinity, i. e., they are never contracted. Note that all other vertices have only incident arcs labeled L in $\boldsymbol{G}$. As a result, shortcuts only span arcs within one modal network. Hence, we neither obtain inhomogeneous travel time functions nor "mixed" super labels. We set the label of each shortcut arc a' to $L(a)$, where a is an (arbitrary) arc along the path represented by a'.

To determine if a shortcut $a' = (u, w)$ is required (when contracting a vertex v), we restrict the witness search to the modal subnetwork G_L of v. Restricting witness search does never yield incorrect query results: Only *unnecessary* shortcuts might be inserted, but no required shortcuts are *omitted*. In fact, this is a common technique to accelerate CH preprocessing [GSSV12]. Note that broadening the witness search beyond network boundaries is prohibitive in our case: It may find a shorter u–w-path using parts of other modal networks. However, such a path is not necessarily a witness if one of these other modes is forbidden during the query. Thus, we must not take it into account to determine if a' can be omitted.

Our preprocessing results in a partial hierarchy for each modal network of $\boldsymbol{G}$. Its transfer vertices are not contracted, thus, stay at the "top" of the hierarchy. Recall that we call the subgraph induced by all vertices v with $\text{rank}(v) = \infty$ the *core*. Because of the added shortcuts, the shortest path between every pair of core vertices is also fully contained in the core, i. e., the core is an overlay graph of $\boldsymbol{G}$. As a result, we achieve independence from the automaton $\mathcal{A}$ during preprocessing.

A Practical Variant. Recall that contraction is independent for every modal network of $\boldsymbol{G}$: We may use any combination of partial, full or no contraction. Our practical



(a) Input graph. (b) Graph after contraction.

Figure 5.6. Contracting only route vertices in the realistic time-dependent model [PSWZ08]. The top row are stop vertices, and the bottom row are route vertices contracted in the order depicted by their labels. Grey arcs represent added shortcuts. Note that these shortcuts are required as they incorporate different transfer times (for entering and exiting vehicles at different stops).

practical variant only contracts time-independent modal networks, that is, the road networks. Contracting the time-dependent networks is much less effective. Recall that we do not contract stop vertices as they have incident link arcs. Applying contraction only on the non-stop vertices, however, yields too many shortcuts (see Figure 5.6 and [Gei10]). It also hides information encoded in the timetable model (such as routes), further complicating query algorithms [BDGM09].

Query. Our query algorithm combines the concept of a multimodal Dijkstra algorithm with unimodal Contraction Hierarchies. Let $s, t \in \boldsymbol{V}$ be source and target vertices and $\mathcal{A}$ a finite automaton with respect to LCSPP-MS. Our query algorithm works as follows.

First, it initializes distance values for all pairs of $(u, q) \in \boldsymbol{V} \times \mathcal{A}$ with infinity. It now runs a bidirectional Dijkstra search from s and t. Each search runs independently and maintains priority queues $\overrightarrow{Q}$ and $\overleftarrow{Q}$ of tuples (u, q) where $u \in \boldsymbol{V}$ and $q \in \mathcal{A}$. We explain the algorithm for the forward search; the backward search works analogously. The queue $\overrightarrow{Q}$ is ordered by distance and initialized with (s, q) for all initial states q in $\mathcal{A}$ (the backward queue is initialized with respect to final states). Whenever the algorithm extracts a tuple (u, q) from Q, it scans all arcs $a = (u, v)$ in $\boldsymbol{G}$. For each such arc, it looks at all states q' in $\mathcal{A}$ that can be reached from q by $L(A)$. For every such pair (v, q') the algorithm checks whether its distance is improved and updates the queue, if necessary.

To incorporate the data from preprocessing, we consider the graph $\boldsymbol{G}$, augmented by all shortcuts computed during preprocessing. The algorithm then runs the aforementioned method, but when scanning arcs from a vertex u, the forward search only looks at arcs (u, v) where $\text{rank}(u) \geq \text{rank}(v)$. Similarly, the backward search only looks at arcs (u, v) where $\text{rank}(u) \geq \text{rank}(v)$. Note that by these means the algorithm automatically searches inside the core whenever it reaches the "top" of the

hierarchy. Thereby, it never reinitializes any data structures when entering the core like it is typically the case for core-based algorithms, such as Core-ALT [DSSW09a]. The stopping criterion carries over from basic CH: A search stops as soon as its minimum key in the priority queue exceeds the best tentative distance value μ. We also use stall-on-demand [GSSV12], however, only on the component.

Intuitively, the search can be interpreted as follows. It simultaneously searches upwards in those hierarchies of the modal networks that are either marked as initial or as final in the automaton $\mathcal{A}$. As soon as it hits the "top" of the hierarchy, it operates on the common core. Because it always finds correct shortest paths between core vertices in *any* modal network, the algorithm supports *arbitrary* automata (with respect to LCSPP-MS) as query input. Note that our algorithm implicitly computes *local* queries which use only one of the networks. It makes the use of a separate algorithm for local queries, as in [DPW09a], unnecessary.

Handling Time-Dependency. Some of the networks in G are time-dependent. Weights of time-dependent arcs (u, v) are evaluated for a departure time τ. However, running a reverse search on a time-dependent network is non-trivial, since the arrival time at the target vertex is not known in advance. Several approaches, such as using the lower-bound graph for the reverse search, exist [DN08, BGNS10], but they complicate the query algorithm. Recall that in our practical variant we do not contract any of the time-dependent networks, hence, no time-dependent arcs are contained in the component. This makes backward search on the component easy for us. We discuss search on the core in the next section.

5.3.4. Improvements

We now present improvements to our algorithm, some of which also apply to Contraction Hierarchies.

Average Node Degree. Recall that whenever the algorithm contracts a modal network, it never contracts transfer vertices, even if they were of low importance in the context of that network. As a result, the number of added shortcuts may increase significantly. Thus, the algorithm stops the contraction process as soon as the *average vertex degree* in the core exceeds a value α. By varying α, we trade the number of core vertices and the number of core arcs: Higher values of α produce a smaller core but with more shortcut arcs. We evaluated a good value of α experimentally.

Arc Ordering. Due to the higher average vertex degree compared to unimodal Contraction Hierarchies, the query algorithm has to look at more arcs. Thus, we improve performance of iterating over incident arcs of a vertex u by *reordering* them locally at u: We first arrange all outgoing arcs, followed by all bidirected arcs and,

finally, all incoming arcs. By these means, the forward respective backward search only needs to look at their relevant subsets of arcs at u. The same optimization is applied to the stalling routine. Preliminary experiments revealed that arc reordering improves query performance by up to 21 %.

Vertex Ordering. To improve the cache hit rate of the query algorithm, we also reorder vertices such that adjacent vertices are stored consecutively with high probability. We use a DFS-like algorithm to determine the ordering [DGNW13]. Because most of the running time is spent on the core, we also move core vertices to the front. This improves query performance up to a factor of two.

Core Pruning. Recall that a search stops as soon as its minimum key from the priority queue exceeds the best tentative distance value μ. This is conservative, but necessary for CH (and also UCCH) to be correct. However, UCCH spends a large fraction of the search inside the core. We can easily expand road and transfer arcs both forward and backward, but because of the conservative stopping criterion, many core vertices are scanned twice. To reduce this amount, the algorithm does not scan arcs of core vertices u, where u has been scanned by both searches and did not improve μ. A path through u is provably not optimal. This increases performance by up to 47 %.

Another alternative is not applying bidirectional search on the core at all. The forward search continues regularly, while the backward search does not scan arcs incident to core vertices. This approach turns out most effective with a performance increase by a factor of two.

State Pruning. Recall that our query algorithm maintains distances for *pairs* (u, q) where $u \in \boldsymbol{V}$ and $q \in \mathcal{A}$. Thus, whenever it scans an arc $(u, v) \in \boldsymbol{A}$ resulting in some state $q \in \mathcal{A}$, it updates the distance value of (v, q) only if it is improved. It is discarded (or pruned) otherwise. However, we can even make use of a stronger *state pruning* rule: Let q_i and q_j be two states in $\mathcal{A}$. We say that q_i *dominates* q_j if and only if the language $L_{\mathcal{A}}(q_j)$ accepted by $\mathcal{A}$ with modified initial state q_j is a subset of the language $L_{\mathcal{A}}(q_i)$ accepted by $\mathcal{A}$ with modified initial state q_i. In other words, any feasible mode sequence beginning with q_j is also feasible when starting with q_i. As a consequence, when the algorithm is about to update a pair (u, q_j), it can additionally prune (u, q_j) if there exists a state q_i that dominates q_j and where (u, q_i) has smaller distance: Any shortest path from u is provably not using (u, q_j).

As an example, consider the first automaton in Figure 5.7a on Page 142. Let its states be denoted by $\{q_0, q_1, q_2\}$, from left to right. Here, q_0 dominates q_2 with respect to our definition: Any foot path beginning at state q_2 is also a feasible (foot) path beginning at state q_0. Therefore, any pair (u, q_2) can be pruned if (u, q_0) has a smaller distance value than (u, q_2). State pruning improves performance by ≈ 10 %.

State-Independent Search in Component. We use automata to model sequence constraints, however, by definition their state may only change when traversing arcs labeled `link`. In particular, when searching inside the component, there is never a state transition (recall that all link arcs are inside the core). Thus, we use the automaton only on the core. The algorithm starts with a regular (unimodal) CH query. Whenever it is about to insert a core node u into the priority queue for the first time on a branch of the shortest path tree, it creates labels (u, q) for all initial (final) states $q \in \mathcal{A}$ (regarding forward/backward search). Because the amount of scanned vertices in the component is small on average compared to the total search space, we do not observe a gain in running time. However, on large networks with complicated query automata we save several gigabytes of memory during query by keeping only one distance value for each component vertex. Recall that component vertices constitute the major fraction of the graph.

Parallelization. In general, the multimodal graph G is composed of more than one contractable modal subnetwork, for instance foot and car. In this case, we have to run the aforementioned unimodal CH query on every component individually. Because these queries are independent from each other, we may parallelize them quite easily. In a first phase, we allocate one thread for every contracted network which then runs the unimodal CH query algorithm on its respective component until it hits the core. In the second phase, the threads are synchronized, and the algorithm continues the search on the core sequentially. Note that we only need to run the first phase on those components that are represented by an initial or final state in the input automaton $\mathcal{A}$.

Combining all improvements yields a speedup of up to factor 4.9.

5.3.5. Experiments

This section presents an extensive experimental study of our algorithm introduced in Section 5.3.3 and compares it to existing approaches, such as Access-Node Routing [DPW09a].

We conducted our experiments on one core of an Intel Xeon E5430 processor running SUSE Linux 11.1. It is clocked at 2.66 GHz, has 32 GiB of memory and 12 MiB of L2 cache. The program was compiled with GCC 4.5, using optimization level 3. Our implementation is written in C++ using the STL and Boost at some points. As a priority queue we use a 4-ary heap.

Inputs. We assembled a total of six multimodal networks where two are imported from [DPW09a]. Their size figures are reported in Table 5.1.

- For `ny-road-rail`, we combine New York's foot network with the public transit network operated by the New York Metropolitan Transit Authority [Met66]. We

Table 5.1. Comparing size figures of our input instances. The column "col." indicates whether we use the Coloring Model (see Section 4.3.4) for the railway subnetwork. The bottom two instances are taken from [DPW09a].

	Public Transit			Road		
Network	Stops	Connections	Col.	Vertices	Arcs	Density
ny-road-rail	16 897	2 054 896	•	579 849	1 527 594	1 : 56
de-road-rail	6 822	489 801	•	5 055 680	12 378 224	1 : 749
europe-road-rail	30 517	1 621 111	•	30 202 516	72 586 158	1 : 1 133
wo-road-rail-flight	31 689	1 649 371	•	50 139 663	124 625 598	1 : 1 846
de-road-rail(long)	498	16 450	○	5 055 680	12 378 224	1 : 10 711
wo-road-flight	1 172	28 260	○	50 139 663	124 625 598	1 : 139 277

link bus and subway stops to road intersections that are no more than 500 m apart.

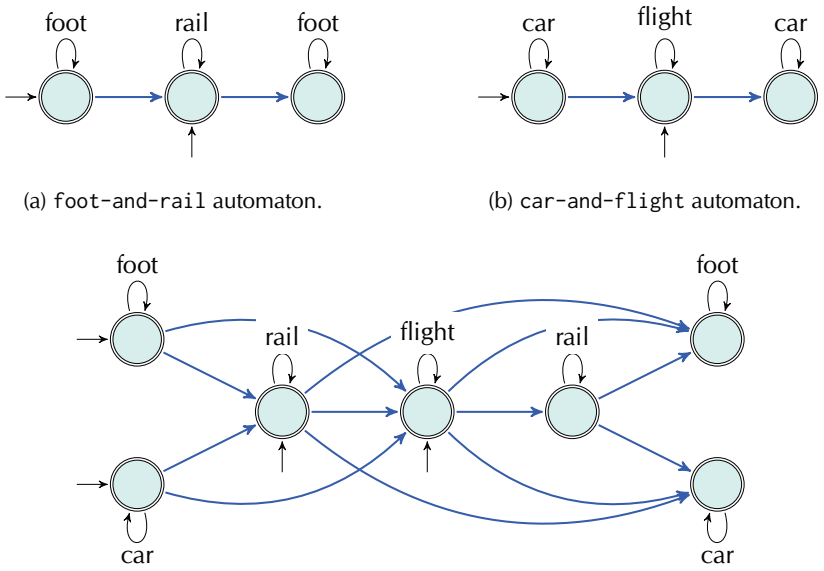
- The de-road-rail network combines the pedestrian and railway networks of Germany. The railway network is extracted from the timetable of the winter period 2000/01. It includes short and long distance trains (which are operated by Deutsche Bahn). We link stations using a radius of 500 m.

- The europe-road-rail network combines the road (as in car) and railway networks of Western Europe. The railway network is extracted from the timetable of the winter period 1996/97 and stations are linked within a radius of 5 km.

- The wo-road-rail-flight network is a combination of the road networks of North America and Western Europe with the railway network of Western Europe and the flight networks of Star Alliance and One World. The flight networks are extracted from the winter timetable 2008. As radius we use 5 km.

- Both de-road-rail(long) and wo-road-flight stem from [DPW09a]. The first combines the road network of Germany with all long-distance trains from the railway network of Germany (which is a subset of the de-road-rail instance). The latter combines the road networks of Europe and North America with the flight networks of Star Alliance and One World. It is a subset of the wo-road-rail-flight network.

The data of the Western European and North American road networks (thereby also Germany and New York) was kindly provided to us by PTV AG [PTV79] for scientific use. The timetable data of New York is publicly available through General Transit Feeds [Gen10], while the data of the German and European railway networks was kindly provided by HaCon [HaC84]. Unfortunately, the New York timetable did



(a) `foot-and-rail` automaton.

(b) `car-and-flight` automaton.

(c) `hierarchical` automaton.

Figure 5.7. Finite automata representing mode sequence constraints that are used in our Experiments. The blue arcs are labeled "link".

not contain any foot path data for transfers. Thus, we generated artificial foot paths with the heuristic presented in Section 4.3.5.

Our instances have varying fractional size of their public transit parts. We call the fraction of linked vertices in a subgraph *density* (see last column of Table 5.1). Our densest network is `ny-road-rail`, which also has the highest number of connections. On the other hand, `de-road-rail(long)` and `wo-road-flight` are rather sparse. However, we include them to compare our algorithm to Access Node Routing (ANR). Also note that for this reason we do not use the improved coloring model from Section 4.3.4 (but the realistic time-dependent model, cf. Section 4.3.3) on these two instances.

We use the following automata as query input. The `foot-and-rail` automaton (Figure 5.7a) allows either walking, or walking, taking the railway network and walking again. Similarly, the `car-and-rail` automaton uses the road network instead of walking, while the `car-and-flight` (Figure 5.7b) automaton uses the flight network instead of the railway network. The `hierarchical` automaton (Figure 5.7c) is our most complicated automaton. It hierarchically combines road, railways and flights (in this order). All modal sequences are possible, except of going up in the hierarchy after once stepping down. For example, if one takes a train after a flight, it is impossible to take another flight. Finally, the `everything` automaton allows arbitrary modal sequences in any order.

Methodology. We evaluate both preprocessing and query performance. We always compute the contraction order online during preprocessing according to the aggressive variant from [GSSV12]. We report the time and the amount of computed auxiliary data. Queries are generated with source, target vertices and departure times uniformly picked at random. For Dijkstra's algorithm we ran 1 000 queries, while for UCCH we run a superset of 100 000 queries.

We report the average number of: (1) extracted vertices in the implicit product graph from the priority queue (scanned vertices), (2) priority queue update operations (relaxed arcs), (3) touched arcs, (4) the average query time, and (5) the speedup over Dijkstra's algorithm. Note that we only report the time to compute the length of the shortest path. Unpacking of shortcuts can be done efficiently in less than a millisecond [GSSD08].

Evaluating Average Core Degree Limit

The first experiment evaluates preprocessing and query performance with varying average core degree. We abort contraction as soon as the average vertex degree in the core exceeds a limit α (cf. Section 5.3.4). In our implementation we compute the average vertex degree as follows. We divide the number of arcs by the number of vertices in the graph data structure. Note that we use an adjacency array data structure with *arc compression* [Del09]: Whenever there are arcs $a = (u, v)$ and $a' = (v, u)$ where $\text{len}(a) = \text{len}(a')$, it combines both arcs in a single entry at u and v. As a result, the number we report may be smaller than the true average degree (at most by a factor of two). This is, however, irrelevant for the result of this experiment.

Table 5.2 shows preprocessing and query figures on `de-road-rail`. We use an automaton that does not use public transit arcs. With higher values of α more vertices are contracted, resulting in higher preprocessing time and more shortcuts (we report them as a fraction of the input's size). At the same time, less vertices (but with higher degree) remain in the core. Setting $\alpha = \infty$ is infeasible. The amount of shortcuts explodes, and preprocessing does not finish within reasonable time. Interestingly, the query time decreases (with smaller core size) up to $\alpha \approx 25$ and then increases again. Though we scan less vertices, the increase in shortcuts results in more touched arcs during query, that is, arcs the algorithm has to iterate when scanning a vertex.

We conclude that for `de-road-rail` the tradeoff between number of core vertices and added shortcut arcs is optimal for $\alpha = 25$. Hence, we use this value in subsequent experiments. Accordingly, we determined α for all other instances.

Comparison to Unimodal CH. In Table 5.2 we also compare UCCH to CH when run on the unimodal road network. We observe that computing a full hierarchy results in queries that are faster by a factor of 11.2. Since UCCH does not compute a full hierarchy by design, we evaluate two partial Contraction Hierarchies: The first stops when the core reaches a size of 10 635—equivalent to the optimal core size

Table 5.2. Performance of UCCH, partial CH, and CH on `de-road-rail` with varying average core degree limit. For queries we use the `foot` automaton.

Preprocessing				Query			
Avg. Core-Degree	Core-Vertices	Shortcut-Arcs	Time [min]	Scanned Vertices	Relaxed Arcs	Touched Arcs	Time [ms]
UCCH:							
10	30 908	42.3 %	6	15 531	27 506	155 776	5.85
15	16 003	43.1 %	7	8 090	16 844	121 631	3.11
20	12 239	43.7 %	9	6 240	14 425	124 201	2.82
25	10 635	44.2 %	10	5 465	13 687	135 151	2.80
30	9 742	44.7 %	12	5 049	13 486	148 735	2.96
35	9 171	45.1 %	14	4 794	13 598	163 376	3.15
40	8 788	45.4 %	15	4 628	13 787	179 483	3.38
PARTIAL CH:							
13	10 635	41.7 %	6	5 567	11 402	71 860	1.93
15	6 750	41.8 %	7	3 636	7 970	53 655	1.37
CH:							
—	0	41.8 %	9	677	1 290	11 434	0.25

of UCCH. We observe a query performance almost comparable to UCCH (slightly faster by 45 %). The second partial hierarchy stops with a core size of 6 750 which is equal to the number of transfer vertices in the network (i. e., the smallest possible core size on this instance for UCCH). Here, CH is a factor of 2 faster than UCCH. Recall that UCCH must not contract transfer vertices. In road networks these are usually "unimportant": Long-range shortest paths do not often pass railway stations or bus stops in general, which explains that UCCH's hierarchy is less pronounced. However, for *multimodal* queries transfer vertices are indeed very important, as they constitute the interchange points between different networks. To enable arbitrary automata during query, we overestimate their importance by not contracting them at all, which is reflected by the (relatively small) difference in performance compared to CH.

Preprocessing

Table 5.3 shows preprocessing figures for UCCH on all our instances. Besides the average degree we evaluate the core in terms of total and fractional number of core vertices and the amount of added shortcuts. Added shortcuts are reported as percentage of all road arcs and in total MiB.

We observe that the preprocessing effort correlates with the graph size. On the small

Table 5.3. Preprocessing figures for UCCH and ANR on the road subnetwork. Figures for the latter are taken from [DPW09a]. We scale the preprocessing time with respect to running time figures compared to Dijkstra's algorithm.

		UCCH					ANR	
	Avg. Core-	Core Vertices		Shortcuts		Time	Space	Time
Network	Deg.	Total	Ratio	Percent	[MiB]	[min]	[MiB]	[min]
ny-road-rail	8	11 057	1:52	48.3 %	8	< 1	—	—
de-road-rail	25	10 635	1:475	44.2 %	63	10	—	—
europe-road-rail	25	39 665	1:761	39.0 %	324	38	—	—
wo-road-rail-flight	30	38 610	1:1 298	39.1 %	558	87	—	—
de-road-rail(long)	35	996	1:5 075	42.3 %	60	10	504	26
wo-road-flight	35	727	1:68 967	38.0 %	542	78	14 050	184

ny-road-rail instance it takes less than a minute and produces 8 MiB of data. On our largest instance, wo-road-rail-flight, we need 1.5 hours and produce 558 MiB of data. Because the size of the core depends on the size of the public transit network, we obtain a much higher ratio of core vertices on ny-road-rail (1 : 52) than we do, for example, on wo-road-rail-flight (1 : 1 298).

Comparing the preprocessing effort of UCCH to scaled figures of Access-Node Routing (ANR) [DPW09a], we observe that UCCH is more than twice as fast and produces significantly less amount of data: On de-road-rail(long) by a factor of 8.4, while on wo-road-flight, ANR requires 14 GiB of space. Here, UCCH only uses 542 MiB, a factor of 26. Concluding, UCCH outperforms ANR in terms of preprocessing space and time.

Query Performance

In this experiment we evaluate the query performance of UCCH and compare it to Dijkstra's algorithm and Access-Node Routing (ANR) where applicable. Figures are presented in Table 5.4. We observe that we achieve speedups of several orders of magnitude over Dijkstra's algorithm, depending on the instance.

Generally, UCCH's speedup over Dijkstra's algorithm correlates with the ratio of core vertices present after preprocessing (thus, indirectly with the density of transfer vertices): The sparser our networks are interconnected, the higher the speedups we achieve. On our densest network, ny-road-rail, we obtain a speedup of 17, while on wo-road-flight we achieve query times of less than a millisecond—a speedup of over 50 540. Note that most of the time is spent inside the core (particularly, in the public transit network), which we do not accelerate. (A detailed query time distribution analysis follows later in this section.) Comparing UCCH with ANR, we observe that query times are in the same order of magnitude, UCCH's being slightly

faster. Note that we achieve these running times with significantly less preprocessing effort.

Improvements

In Table 5.5 we report figures for the improvements to UCCH which we described in Section 5.3.4. The table is divided in two parts. The upper part addresses unimodal improvements that are also applicable to (partial) CH. Therefore, we evaluate them using the `car` automaton. For our two biggest networks, we provide the number of scanned vertices and the query time for several combinations of improvements. The first row (none) reports results for the basic version of UCCH. The other rows use: Reordered vertices (rv), reordered arcs (ra), improved bidirectional search on the core (bi), and unidirectional search on the core (fo), that is, no backward search is performed on the core. Combining these techniques, we obtain a speedup of up to a factor of 4.8.

The lower part of Table 5.5 is dedicated to improvements that are specific for UCCH. We evaluate them using the `car-and-rail` automaton. We provide numbers for state-independent search on the component (si) and state-pruning (sp). Note that these figures already include the previous improvements. Interestingly, using state-independent search results in slightly worse query times of about 5 %. However, we reduce the memory footprint of the algorithm by a significant amount (up to several Gigabytes) since we store distance values only once per component vertex.

Note that from the number of scanned vertices we can deduce which of the improvements impact cache efficiency and which impact the search space.

In-Depth Analysis of Query Performance

Table 5.6 reports in-depth figures for the UCCH query including all (reasonable) improvements from the previous section. We see that a large fraction of the query is spent on the public transit part of the multimodal network: Up to 65 % of the scanned vertices and also up to 65 % of query time. Recall that we do not further accelerate the search on the core.

Interestingly, UCCH is slightly faster (up to a factor of 2.6) on the public transit subnetworks when compared to Dijkstra's algorithm. UCCH scans fewer vertices in total, which helps cache performance on the public transit part.

When we compare the time spent on the road network (component and core) of `de-road-rail` with the figures of Table 5.2 (where we use the same instance but with the smaller `foot` automaton), we observe that the `foot-and-rail` automaton yields a factor 1.8 slowdown. The reason is that the `foot-and-rail` automaton actually has two "foot states" (cf. Figure 5.7a) and, thus, has to do twice the work on the road subnetwork. Note that the number 1.8 (instead of exactly 2) stems from the fact that we apply state pruning.

Table 5.4. Query performance of UCCH compared to multimodal Dijkstra ANR. Figures for the latter are taken from [DPW09a]. We scale the running time with respect to Dijkstra's algorithm.

		Dijkstra		**ANR**			**UCCH**		
Network	Automaton	Scanned Vertices	Time [ms]	Scanned Vertices	Time [ms]	Speed-up	Scanned Vertices	Time [ms]	Speed-up
ny-road-rail	foot-and-rail	404 816	226	—	—	—	25 525	13.61	17
de-road-rail	foot-and-rail	2 611 054	2 005	—	—	—	18 275	12.78	157
europe-road-rail	car-and-rail	30 021 567	23 993	—	—	—	90 579	53.78	446
wo-road-rail-flight	car-and-flight	36 053 717	33 692	—	—	—	42 056	26.72	1 260
wo-road-rail-flight	hierarchical	36 124 105	35 261	—	—	—	126 072	70.52	500
wo-road-rail-flight	everything	25 267 202	23 972	—	—	—	71 389	50.77	472
de-road-rail(long)	foot-and-rail	2 735 426	2 075	13 524	3.45	602	12 509	3.13	663
wo-road-flight	car-and-flight	36 582 904	33 862	4 200	1.07	31 551	1 647	0.67	50 540

Table 5.5. Detailed analysis of the impact on query performance by our improvements (cf. Section 5.3.4).

Network	Automaton	*rv*	*ra*	*bj*	*fo*	*si*	*sp*	Scanned Vertices	Time [ms]	Spd.-up
europe-road-rail	car	○	○	○	○	○	○	48 488	69.93	—
europe-road-rail	car	●	○	○	○	○	○	48 488	35.11	2.00
europe-road-rail	car	●	●	○	○	○	○	48 488	29.38	2.38
europe-road-rail	car	●	●	●	○	○	○	31 628	20.02	3.49
europe-road-rail	car	●	●	○	●	○	○	24 297	14.57	4.80
wo-road-rail-flight	car	○	○	○	○	○	○	35 539	54.42	—
wo-road-rail-flight	car	●	○	○	○	○	○	35 539	27.93	1.95
wo-road-rail-flight	car	●	●	○	○	○	○	35 539	23.18	2.35
wo-road-rail-flight	car	●	●	●	○	○	○	29 695	19.84	2.74
wo-road-rail-flight	car	●	●	○	●	○	○	17 862	11.50	4.73
europe-road-rail	car-and-rail	●	●	○	●	○	○	95 095	57.23	—
europe-road-rail	car-and-rail	●	●	○	●	●	○	95 024	60.12	0.95
europe-road-rail	car-and-rail	●	●	○	●	○	●	89 770	51.72	1.11
europe-road-rail	car-and-rail	●	●	○	●	●	●	89 699	54.45	1.05
wo-road-rail-flight	car-and-rail	●	●	○	●	○	○	72 997	46.73	—
wo-road-rail-flight	car-and-rail	●	●	○	●	●	○	72 895	49.09	0.95
wo-road-rail-flight	car-and-rail	●	●	○	●	○	●	69 627	42.35	1.10
wo-road-rail-flight	car-and-rail	●	●	○	●	●	●	69 525	44.51	1.05

5.3.6. Conclusion

In section we introduced UCCH: A fast multimodal speedup technique that handles arbitrary modal sequence constraints at *query time*—a problem considered challenging before. Besides not determining the modal constraints during preprocessing, its advantages are small space overhead, fast preprocessing time and the ability to implicitly handle local queries without the need for a separate algorithm. Its preprocessing can handle huge intercontinental networks with many more stations and airports than those of previous multimodal techniques.

For future work we are interested in augmenting our approach to more general scenarios such as profile or multicriteria queries. We also like to further accelerate search on the uncontracted core—especially on the rail networks. Moreover, we are interested to improve the contraction order. In particular, we like to use ideas from [DPW09a] to enable contraction of some transfer vertices in order to achieve better results, especially on more densely interlinked networks.

Table 5.6. In-depth analysis of UCCH's query time. We report the distribution of query time among the particular subnetworks and compare it to Dijkstra's algorithm.

	Dijkstra's Algorithm		UCCH		
Subnetwork	Scanned Vertices	Time [ms]	Scanned Vertices	Time [ms]	Speed-up
`ny-road-rail` on `foot-and-rail`:					
road component	—	—	203	≈ 0.0	—
road core	389 578	215.5	9 944	4.8	45
rail	15 238	10.5	15 238	8.8	1.2
`de-road-rail` on `foot-and-rail`:					
road component	—	—	188	≈ 0.0	—
road core	2 599 251	1 988.4	6 314	5.0	397
rail	11 803	16.6	11 803	7.8	2.1
`europe-road-rail` on `car-and-rail`:					
road component	—	—	213	≈ 0.0	—
road core	29 973 817	23 933.3	43 017	24.4	982
rail	47 750	59.7	47 750	29.4	2.0
`wo-road-rail-flight` on `hierarchical`:					
road component	—	—	301	≈ 0.0	—
road core	36 047 522	35 169.3	49 944	30.6	1 149
rail	75 682	89.9	75 682	39.2	2.3
flight	902	1.8	902	0.7	2.6

5.4. Multicriteria Multimodal Route Planning

In the previous section we considered the label-constrained shortest path problem with mode sequence constraints (LCSPP-MS) and presented an algorithm (UCCH) to quickly compute queries obeying mode sequence constraints that are given as a user input. Now, recall for a moment that the motivation for LCSPP(-MS) was to compute journeys, that utilize a *reasonable* sequence of transportation modes. For example, reconsidering Figure 5.1, the automaton shown in Figure 5.8 ensures that one of the desired solutions from Figure 5.1b is computed.

While this may seem a fine approach to the multimodal problem at first, computing label-constrained shortest paths has two disadvantages. First, the output of the query algorithm (such as UCCH's) is a *single* journey that, albeit obeying the label constraints, is the one that arrives at the target earliest. Consequently, one of the options from Figure 5.1 will be hidden from the user, even though, the (user-

specified) automaton encourages both. Second, LCSPP requires the *user* to specify mode constraints. While this may be useful to exclude certain transportation mode sequences, it actually exposes a fundamental conceptual problem with label-constrained optimization: It essentially relies on the user to know their (modal) options *before* planning the journey.

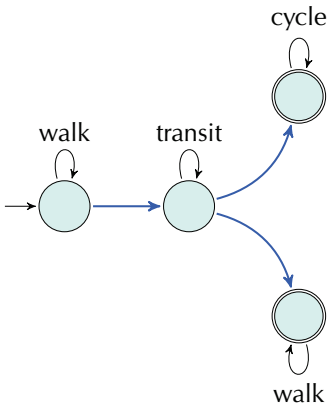


Figure 5.8. Mode sequence constraints corresponding to the journeys in Figure 5.1b.

Given the limitation of label-constrained optimization, we revisit the problem of computing multimodal journeys by combining multimodal route planning with multicriteria optimization in this section. The idea is to obtain a *diverse* (with respect to the available transportation modes) and *concise* set of journeys, from which the user can then *choose* their favorite. We argue that users optimize—besides arrival time—for each transportation mode a mode-dependent *convenience* criterion, such as the number of transfers for public transit, or the total walking duration for walking. Computing Pareto sets then produces solutions for various trade offs in these criteria. For example in Figure 5.1b, the journey that uses walking as its last leg may have an earlier arrival time, however, at the cost of more walking. Eventually, both solutions are Pareto-optimal. Note that the earliest arrival journey from Figure 5.1a is Pareto-optimal, as well. In fact, it is the best solution regarding the criterion arrival time and is, therefore, presented to the user as a third alternative.

Unfortunately, with increasing number of optimization criteria, the size of the Pareto sets grows to the point that there are too many solutions to be presented to the user. Moreover, many of them have insignificant tradeoffs in their criteria, such as saving one minute in walking at the cost of arriving hours later. (An example is presented in Figure 5.13.) To identify *significant* journeys of the Pareto set we propose to use fuzzy logic [Zad88, Zad65] in order to rank them in a postprocessing step [FA04]. This postprocessing step is not only quick but can also incorporate user-dependent preferences on the "fuzziness" of each optimization criterion.

To compute exact Pareto sets, we propose two approaches. The first (MLC) extends the Multi-Label-Correcting algorithm [MSWZ07] (also see Section 4.4.2) to the multimodal scenario. The second, called MCR, is based on RAPTOR (see Section 4.6) and augments the round-based paradigm to incorporate further (unrestricted) modes of transportation. We also propose to combine ideas from Contraction Hierarchies [GSSV12] and UCCH (see Section 5.3) with both MLC and MCR, in order to accelerate queries. Unfortunately, even with these optimizations, queries take several seconds. Therefore, we further present different heuristics (still multicriteria), which aim to not compute those (insignificant) journeys that would be filtered during postprocessing. With these heuristics, we can compute on the full metropolitan

network of London that includes all public transit, rental bicycles, walking, and taxis, solutions of good quality in below one second. For a restricted scenario that excludes taxi, journeys of excellent quality can be computed in under 500 ms time on average, which is fast enough for server applications.

Overview. This section is organized as follows. First, we formally define the problem of multimodal multicriteria optimization as well as the considered criteria in Section 5.4.1. We then show how we use fuzzy logic to relax domination in order to rank the journeys in Section 5.4.2. Section 5.4.3 presents two algorithms that compute exact Pareto sets of multimodal journeys for a restricted problem that only considers public transit and walking: MLC and MCR. Thereby, the latter extends RAPTOR (cf. Section 4.6) to this setting. We then show in Section 5.4.4 how we combine UCCH (cf. Section 5.3) with both MLC and MCR to accelerate queries. Section 5.4.5 extends both algorithms to the scenario that includes taxi and rental bicycles. To reduce the number of solutions maintained during the algorithm, Section 5.4.6 presents several heuristic algorithms (based on MCR). Section 5.4.7 presents a corresponding quality measure (using fuzzy set theory in a consistent way) which enables us to evaluate the (qualitative) performance of the heuristics. Finally, Section 5.4.8 contains an extensive experiments, and Section 5.4.9 some concluding remarks.

References. The content of this section is based on [DDP+13] which appeared at the 12th International Symposium on Experimental Algorithms (2013) and on a prior technical report [DDP+12] which has been published at the Faculty of Informatics of the Karlsruhe Institute of Technology. It is joint work with Daniel Delling, Julian Dibbelt, Renato Werneck, and Dorothea Wagner.

5.4.1. Problem Statement

In the following, a query always takes as input a *source location* s, a *target location* t, and a *departure time* τ. It produces *journeys* that leave s no earlier than τ and arrive at t. Thereby, a *journey* is a valid path in the integrated multimodal transportation network that obeys all timetable constraints.

Criteria. We still have to define *which* journeys the query should return. We argue that users optimize two natural classes of criteria in multimodal networks: arrival time, and mode-dependent "convenience". For our first (simplified) scenario (with public transit, cycling, and walking, but no taxi), we work with three actual criteria. Besides arrival time, we use the number of trips and walking duration as proxies for convenience (for public transit/cycling and walking, respectively). We later add monetary cost for the scenario that includes taxi.

Given this setup, a first natural problem we need to solve is the *full multicriteria problem*, which must return a full (maximal) Pareto set of journeys. We say that a journey J_1 *dominates* J_2 if J_1 is strictly better than J_2 according to at least one criterion and no worse according to all other criteria. A *Pareto set* is a set of pairwise nondominating journeys [MW01, Han79]. If two journeys have equal values in all criteria, we only keep one.

5.4.2. Dominance and Fuzzy Set Theory

Solving the full multicriteria problem, however, can lead to solution sets that are too large for most users (there is an example at the end of this section). Moreover, many solutions provide undesirable tradeoffs, such as journeys that arrive much later to save a few seconds of walking (or walk much longer to save a few seconds in arrival time). Intuitively, most criteria are diffuse to the user, and only large enough differences are significant. Pareto optimality fails to capture this.

To formalize the notion of significance, we propose to *score* the journeys in the Pareto set in a post-processing step using concepts from fuzzy logic [Zad88] (and fuzzy set theory [Zad65]). Loosely speaking, fuzzy logic generalizes Boolean logic to handle (continuous) degrees of truth. For example, the statement "60 and 61 seconds of walking are equal" is false in classical logic, but might be considered "almost true" in fuzzy logic. We define some necessary notion from fuzzy logic in the following.

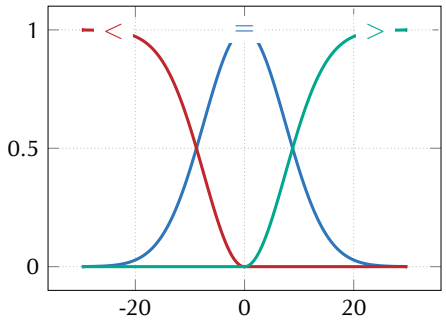
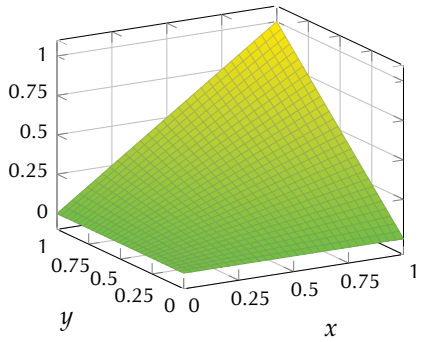


Figure 5.9. Plot of the fuzzy relational operators $\mu_=$ (blue curve), $\mu_>$ (green curve), and $\mu_<$ (red curve) for $\epsilon = 5$ and $\chi = 0.8$.

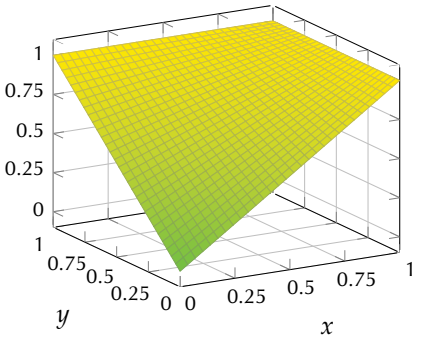
Fuzzy Sets. Formally, a *fuzzy set* is a tuple $\mathcal{S} = (\mathcal{U}, \mu)$, where $\mathcal{U}$ is a set and $\mu\colon \mathcal{U} \to [0,1]$ a *membership function* that defines "how much" each element from $\mathcal{U}$ is contained in $\mathcal{S}$. Mostly, we use μ directly to refer to $\mathcal{S}$. Our application requires fuzzy relational operators $\mu_<$, $\mu_=$, and $\mu_>$ (for "better", "equal" and "worse"). For any values $x, y \in \mathbb{R}$, they are evaluated by $\mu_<(x-y)$, $\mu_>(y-x)$, and $\mu_=(x-y)$. In this section, we use the well-known [Zad88] exponential membership functions for those operators:

$$\mu_=(x) := \exp\left(\frac{\ln(\chi)}{\epsilon^2}x^2\right), \tag{5.2}$$

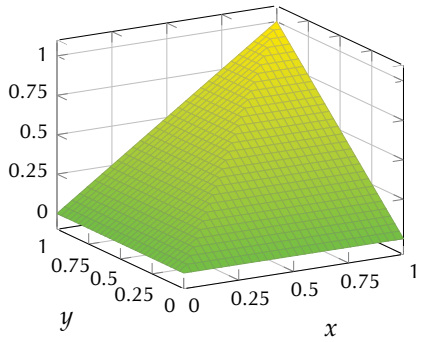
where $0 < \chi < 1$ and $\epsilon > 0$ control the degree of fuzziness (i. e., the "width" of the Gaussian). The other two operators are derived by $\mu_<(x) := 1 - \mu_=(x)$ if $x < 0$ (0 otherwise) and $\mu_> := 1 - \mu_=(x)$ if $x > 0$ (0 otherwise). Figure 5.9 shows plots of $\mu_=$, $\mu_>$, and $\mu_<$ using the exponential membership function from Equation 5.2. The parameters are $\chi = 0.8$ and $\epsilon = 5$, i. e., the Gaussian centered at $x = 0$ has a value of 0.8 for $x = 5$.



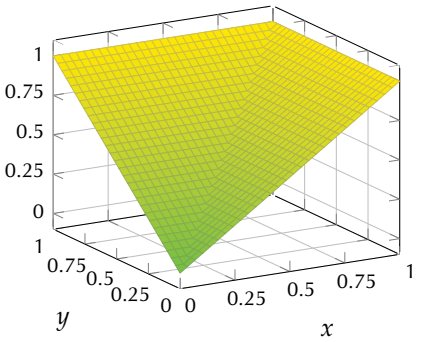
(a) Product norm $T(x,y) = xy$.



(b) Probabilistic sum $S(x,y) = x + y - xy$.



(c) Minimum norm $T(x,y) = \min(x,y)$.



(d) Maximum norm $S(x,y) = \max(x,y)$.

Figure 5.10. Plots for two exemplary t- and s-norms pairs: Product norm, probabilistic sum, minimum norm, and maximum norm.

Norms. Another important concept from fuzzy logic are t-norms and s-norms which correspond to fuzzy binary logical operators. We recap them next. A *triangular norm* (short: *t-norm*) $T\colon [0,1]^2 \to [0,1]$ is a commutative, associative, and monotone (i. e., $a \leq b, x \leq y \Rightarrow T(a,x) \leq T(b,y)$) binary operator to which 1 is the neutral element. If $x, y \in [0,1]$ are fuzzy truth values, $T(x,y)$ is interpreted as a fuzzy conjunction (*and*) of x and y. Given a t-norm T, the *complementary conorm* (or *s-norm*) of T is defined as $S(x,y) := 1 - T(1-x, 1-y)$, which we interpret as a fuzzy disjunction (*or*). Note that the neutral element of S is 0. Two well-known pairs of t- and s-norms are $(\min(x,y), \max(x,y))$, called *minimum/maximum norms*, and $(xy, x+y-xy)$, called *product norm/probabilistic sum*. Plots for both pairs of s- and t-norms are shown in Figure 5.10.

A Notion of Fuzzy Dominance. In the following paragraphs, we recap and (re)derive the concept of *fuzzy dominance* in multicriteria optimization. It has been introduced

by Farina and Amato in [FA04]. Given two journeys J_1 and J_2 with C optimization criteria, we denote by $n_b(J_1, J_2)$ the fuzzy *number of criteria* in which J_1 is better than J_2. More formally,

$$n_b(J_1, J_2) := \sum_{i=1}^{C} \mu^i_<\big(\kappa^i(J_1), \kappa^i(J_2)\big), \tag{5.3}$$

where $\kappa^i(J)$ evaluates the i-th criterion of J and $\mu^i_<$ is the fuzzy less-than operator associated with the i-th criterion. Note that each criterion may use different fuzzy operators to control the amount of fuzzyness for every criterion individually. Analogously, we define $n_e(J_1, J_2)$ for equality and $n_w(J_1, J_2)$ for greater-than (by substituting $\mu^i_<$ in Equation 5.3 by $\mu^i_=$ and $\mu^i_>$, respectively). Now, it holds by definition that $n_b(J_1, J_2) + n_e(J_1, J_2) + n_w(J_1, J_2) = C$ for any pair of journeys J_1 and J_2.

Having established this notion, Pareto dominance can be rephrased as follows. A journey J_1 (strictly) dominates another journey J_2, if and only if $n_e(J_1, J_2) < C$ (they differ at least in one criterion), and $n_w(J_1, J_2) = 0$ (i. e., the journey J_1 is not worse than J_2 in *any* criterion).

k-Dominance. To relax the latter condition, we define the following notion of k-dominance for values of k in the interval $[0, 1] \subset \mathbb{R}$. A journey J_1 *k-dominates* another journey J_2 if and only if

$$n_e(J_1, J_2) < C, \text{ and} \tag{5.4a}$$
$$(1 - k)n_b(J_1, J_2) \geq n_w(J_1, J_2). \tag{5.4b}$$

In other words, the journey J_1 still k-dominates the journey J_2, if it is worse in at least $(1 - k)$-times the number of criteria for which it is better. Setting $k = 1$, we exactly obtain Pareto dominance, since then the left-hand side of Equation 5.4b becomes zero, which leads to $n_w(J_1, J_2) = 0$. We may also say that in this case, journey J_1 "100 %-dominates" journey J_2 (since $k = 1$). In contrast, consider a journey J_1 dominating J_2 for $k = 0$, but *not* for any $k > 0$. For this, it has to hold that $n_b(J_1, J_2) = n_w(J_1, J_2)$. In fact, this may be regarded as the "threshold for domination". Note that for the case that $n_w(J_1, J_2) > n_b(J_1, J_2)$ holds, the journey J_1 never dominates J_2, regardless of the choice of k.

Given the notion of k-dominance, a natural question is the following: For two journeys J_1 and J_2, what is the *largest* value of k, such that J_1 (still) k-dominates journey J_2? Obviously, for such a value of k, Equation 5.4b assumes equality. Hence, we may derive k from Equation 5.4b by transformation:

$$(1 - k)n_b(J_1, J_2) = n_w(J_1, J_2) \tag{5.5a}$$
$$\Leftrightarrow kn_b(J_1, J_2) = n_b(J_1, J_2) - n_w(J_1, J_2) \tag{5.5b}$$
$$\Leftrightarrow k = \frac{n_b(J_1, J_2) - n_w(J_1, J_2)}{n_b(J_1, J_2)}. \tag{5.5c}$$

Note that the last step in the above transformation is only valid if $n_b(J_1, J_2) \neq 0$. However, recall that J_1 may only k-dominate J_2 for a value of $k > 0$, if $n_b(J_1, J_2) > n_w(J_1, J_2)$ holds (cf. Equation 5.4b), and no such $k > 0$ exists, otherwise.

Degrees of Domination. From Equation 5.5c, we may now define a binary function d, which, given two journeys J_1 and J_2, defines the *degree of domination*, i. e., how much J_1 dominates J_2. More formally, we define d by

$$d(J_1, J_2) := \begin{cases} 0 & \text{if } n_b(J_1, J_2) \leq n_w(J_1, J_2), \\ \left(n_b(J_1, J_2) - n_w(J_1, J_2)\right) / n_b(J_1, J_2) & \text{otherwise.} \end{cases} \tag{5.6}$$

Here, $d(J_1, J_2) = 0$ means that J_1 does not dominate J_2, while a value of 1 indicates that J_1 Pareto-dominates J_2. Otherwise, we may also say that J_1 *fuzzy-dominates* J_2 by degree $d(J_1, J_2)$. Note that we never divide by zero, since for $n_b(J_1, J_2) = 0$, the first case in Equation 5.6 (always) applies.

Figure 5.11 shows contour lines for values of d between 0 and 1 for two exemplary criteria: arrival time and walking duration (with fuzziness parameters set as in Section 5.4.8). In the figure we fix the criteria of J_1 to $(0, 0)$. The area right-above each contour line t then contains all journeys J_2 (with respective values for their criteria) which are dominated by J_1 with degree at least t. For example, a journey is still dominated by J_1 with degree 0.4 if it has 10 minutes less walking while arriving 5 minutes later. The corresponding surface plot of Figure 5.11 is shown in Figure 5.12

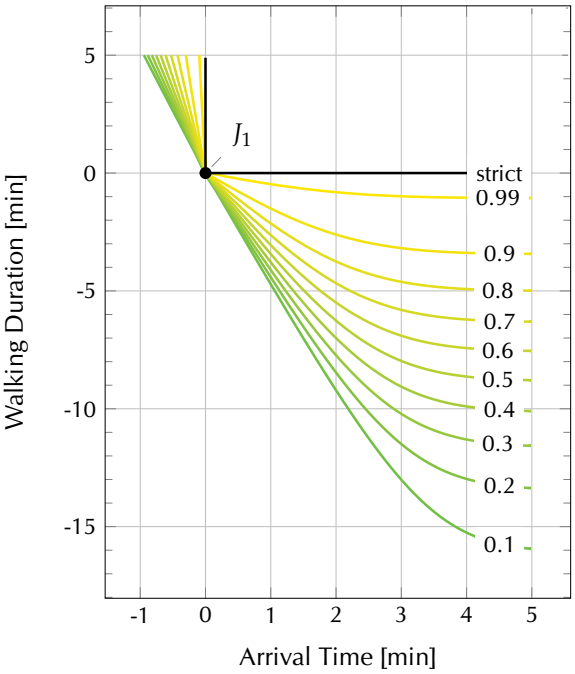


Figure 5.11. Contour lines of the fuzzy dominance function $d(J_1, J_2) = t$ for different values t. The black line marks strict Pareto-dominance.

Scoring Journeys. Now, given any (Pareto) set $\mathcal{J}$ of n journeys $J_1, \ldots, J_n$, we define a *score function* $\mathrm{sc}\colon \mathcal{J} \to [0, 1]$ that computes the degree of domination by the whole set for each individual journey $J \in \mathcal{J}$ (independently). More precisely, $\mathrm{sc}(J) := 1 - S(J_1, \ldots, J_n)$. (We extend S beyond two parameters recursively, i. e, $S(J_1, \ldots, J_n) = S(S(J_1, \ldots, J_{n-1}), J_n)$.) Note that if we set S to be the maximum norm, the score is determined by the (one) journey that dominates J most. On the other hand, with the probabilistic sum the score may be based on several fuzzily dominating journeys.

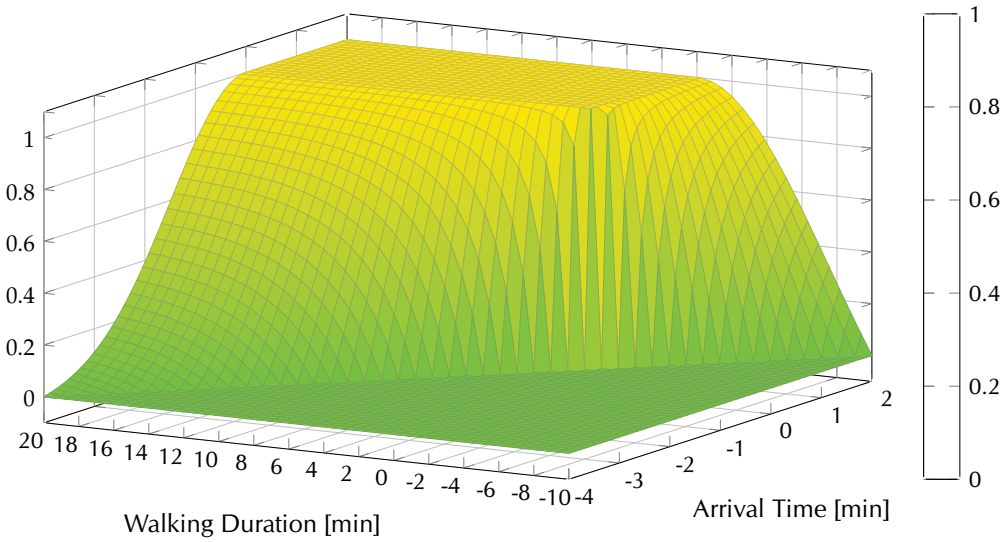


Figure 5.12. Fuzzy dominance function $d(J_1, J_2)$ for journeys J_1 and J_2. We fix J_1 to $(0, 0)$ and vary $J_2 = (x, y)$. The plot shows how much J_1 dominates J_2.

We finally use the score to order the journeys by significance. One may then decide to only show the k journeys with highest score to the user.

Example. Figure 5.13 shows a (quite representative) location-to-location query from William Road (near Warren Street Station) to Caxton Street (near Westminster Abbey) on our London instance using public transit, walking, and taxi with optimization criteria arrival time, number of transfers, walking duration, and cost (in pounds). (More details on the input are found in Section 5.4.8.) The departure time is 4:27 pm. The left figure shows all nondominating journeys of the full Pareto set (there are 65 in total), while the right figure shows the three journeys with highest score from the (same) Pareto set, when our fuzzy dominance approach is used. This example clearly demonstrates that we obtain too many nondominating solutions (left figure), a known problem for multicriteria search. But not only is the number of solutions too high for presentation to a user, in fact, most of the journeys are not meaningful. Some of them take considerable detours (for example north of the source location), just to save some (insignificant) amount of walking. In contrast, our scoring approach by fuzzy domination (right figure) is able to identify the significant solutions in the Pareto set, resulting in three meaningful journeys: One taking taxi the full way (purple), one taking the subway (blue) which is faster at the cost of more walking (black), and one taking the bus (red) which takes longer but with significantly less total walking (4 min instead of 14 min).

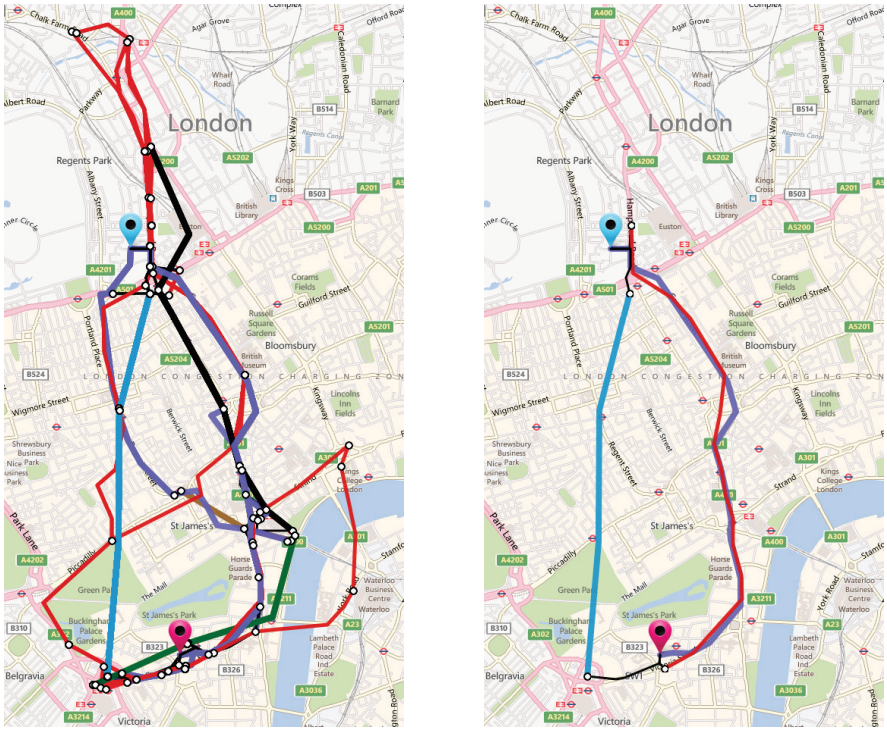


Figure 5.13. Exemplary multicriteria multimodal query on London with criteria arrival time, number of transfers, walking duration, and cost. The left figure shows the full Pareto set (65 journeys), while the right figure shows the three journeys with highest score. Each dot represents a transfer and included transportation modes are walking (thin black), taxi (thick purple), buses (thin red), and tube (other thick colors).

5.4.3. Exact Algorithms

This section considers exact algorithms for the multicriteria multimodal problem to obtain a Pareto set (which may then be used to score its journeys by the fuzzy dominance approach). Recall that for the unimodal, but multicriteria, problem in public transit networks, Section 4.4.2 considered a graph-based solution called Multi-Label-Correcting algorithm (MLC), while in Section 4.6 we proposed RAPTOR, which computes Pareto sets of public transit journeys that include number of transfers as a criterion. In this section, we first build on these two algorithms and describe how they are extended to the multimodal multicriteria scenario. In the subsequent Section 5.4.4 we then describe how ideas from User-Constrained Contraction Hierarchies (UCCH) (cf. Section 5.3) apply to both algorithms.

To simplify the discussion (and notation), we first describe the algorithms in terms of our simplest scenario, considering only the (timetable-based) public transit network and the (unrestricted) walking network. Section 5.4.5 explains how to handle cycling and taxis, which are unrestricted but have special properties.

MLC: Multi-Label-Correcting Algorithm. As we introduced in Section 4.3, traditional solutions to the multicriteria problem on public transit networks typically model the timetable as a graph [BDGM09, DKP12, Gei10, MSWZ07]. A particularly effective approach is the *realistic time-dependent model* [MSWZ07], recapped in Section 4.3.3. Recall that for each stop p, it creates a single *stop vertex* linked by time-independent *transfer arcs* to multiple *route vertices*, one for each route serving p. It also adds *route arcs* between route vertices associated to consecutive stops within the same route. To model the trips along a route, the cost of a route arc is given by a piecewise linear function reflecting the traversal time (including waiting for the next departure).

Given this model, a journey in the public transportation network corresponds to a path in this graph. The *multi-label-correcting* (MLC) [MSWZ07] algorithm uses this to find full Pareto sets for arbitrary criteria that can be modeled as arc costs. MLC extends Dijkstra's algorithm [Dij59] (also see Section 4.4.1) by operating on labels that have multiple values, one per criterion. Each vertex u maintains a *bag* $B(u)$ of nondominated labels. In each iteration, MLC extracts from a priority queue the minimum (in lexicographic order) unprocessed label $L(u)$. For each arc (u, v) out of the associated vertex u, MLC creates a new label $L(v)$ (by extending $L(u)$ in the natural way) and inserts it into $B(v)$; newly-dominated labels (possibly including $L(v)$ itself) are discarded, and the priority queue is updated, if needed. MLC can be sped up with target pruning and by avoiding unnecessary domination checks. See Section 4.4.2 and [MSWZ07] for details.

To solve the multimodal problem, we extend MLC: It suffices to augment its input graph to include the walking network. We combine the original graphs by merging (public transportation) stops and (walking) intersections that share the same location (and keeping all arcs). These *link vertices* are then used to switch between modes of transportation. The MLC query remains essentially unchanged and still processes labels in lexicographic order. Although labels can now be associated to vertices in different networks, they can all share the same priority queue.

MCR: Round-based Algorithm. A drawback of MLC (even restricted to public transit networks) is that it can be quite slow: Unlike Dijkstra's algorithm, MLC may scan the same vertex multiple times (the exact number depends on the criteria being optimized), and domination checks make each such scan quite costly. In the context of unimodal public transit networks, Section 4.6 presented the RAPTOR algorithm, a faster alternative. To better understand how it is extended to multimodal queries, we briefly recap it in the following. For more details on RAPTOR, see Section 4.6.

The simplest version of the algorithm optimizes two criteria: arrival time and number of transfers. Unlike MLC, which searches a graph, RAPTOR uses dynamic programming to operate directly on the timetable. It works in rounds, with round i processing all relevant journeys with exactly $i - 1$ transfers (or, in other words, i trips). It maintains one label per round i and stop p representing the best known

arrival time at p for up to i trips. During round i, the algorithm processes each *route* once. It reads arrival times from round $i-1$ to determine relevant trips (on the route) and updates the labels of round i at every stop along the way. Once all routes are processed, the algorithm considers potential transfers to nearby (predefined) stops in a second phase. Simpler data structures and better locality make RAPTOR an order of magnitude faster than MLC. In Section 4.6.6 we also proposed McRAPTOR, which extends RAPTOR to handle more criteria (besides arrival times and number of transfers). It maintains a *bag* (set) of labels with each stop and round.

Even with multiple modes of transport available, one trip always consists of a single mode. Moreover, switching modes of transports only occurs at a limited set of locations (i. e., those entities such as stops or vertices that are linked). This motivates adapting the round-based paradigm to the multimodal scenario.

We, therefore, propose MCR (*multimodal multicriteria RAPTOR*), which extends McRAPTOR to handle multimodal queries. As in McRAPTOR, each round has several stages. Recall from Section 4.6.1 that stage I initializes labels, stage II processes routes in the public transit network, and stage III considers footpaths. While stage III does not apply in the multimodal scenario, we substitute it by a new stage for each additional (besides public transit) network. Therefore, to enable unrestricted walking, the third stage considers *arbitrary* paths in the unrestricted walking network. We compute them by using MLC (on the walking network). Thereby, MLC extends bags instead of individual labels. To ensure that each label is processed at most once, we keep track of which labels (in a bag) have already been extended. Therefore, during round i, each stage in McRAPTOR reads labels from round $i-1$ (which have been computed in the previous round and are, thus, final) and writes to round i. In contrast, the MLC subroutine may read and write labels of the same round i, if walking is not regarded as a trip. However, in this case, it has to be run as the last stage of round i (similarly to the footpath stage described in Section 4.6.1. Note that additional modal subnetworks can be easily added to MCR (by just adding another stage), and dedicated (sub)algorithms can be used for each. Moreover, any two subnetworks which read labels from round $i-1$ and write to round i can be processed independently and in arbitrary order. This allows easy parallelization of different modal subnetworks.

Finally, to enable queries between *arbitrary* locations (rather than only transfer locations), the initialization routine (before the first round) runs Dijkstra's algorithm on the walking network from the source p_s to determine the fastest walking path to each stop in the public transportation network (and to p_t), thus creating the initial labels used by MCR.

5.4.4. Contracting the Unrestricted Networks

As our experiments will show in Section 5.4.8, the bottleneck of the multimodal algorithms is processing the walking network $G = (V, A)$. We improve performance

by combining MLC (which is run in the walking stage) and UCCH from Section 5.3.

For any journey involving public transit, walking between trips always begins and ends at the restricted set $V_{\texttt{link}} \subset V$ of link vertices. During queries, we must only be able to compute the pairwise distances between these vertices. We therefore use UCCH's preprocessing (cf. Section 5.3.3) to compute a smaller *core graph* that preserves these distances. Recapping it briefly, the algorithm starts from the original graph and iteratively *contracts* [GSSV12] (cf. Section 5.3.1) each vertex in $V \setminus V_{\texttt{link}}$ in the order given by a rank function rank. Each contraction step (temporarily) removes a vertex and adds shortcuts between its uncontracted neighbors to maintain shortest path distances (if necessary). It is usually advantageous to first contract vertices with relatively small degrees that are evenly distributed across the network [GSSV12]. We stop contraction when the average degree in the core graph reaches some threshold (we use 12 in our experiments). See Section 5.3.3 for details.

To run a faster multimodal s–t query (for any vertices $s, t \in V$), we use essentially the same algorithm as before (based on either MLC or RAPTOR), but replacing the full walking network with the (smaller) core graph. Since the source s and the target t may not be in the core, we handle them during initialization. It works on the graph $G^+ = (V, A \cup A^+)$ containing all original arcs A as well as all shortcuts A^+ added during the contraction process. We run upward searches (only following arcs (u, v) such that $\text{rank}(u) > \text{rank}(v)$) in G^+ from s (scanning forward arcs) and t (scanning reverse arcs); they reach all potential entry and exit points of the core, but arcs within the core are not processed. These core vertices (and their respective distances) are used as input to MCR's (or MLC's) standard initialization, which can operate only on the core from this point on.

The main loop works as before, with one minor adjustment. Whenever MLC extracts a label $L(u)$ for a scanned core vertex u, it check if it has been reached by the reverse search during initialization. If so, it creates a temporary label $L'(t)$ by extending $L(u)$ with the (already computed) walking path to t and adds it to $B(t)$ if needed. MCR is adjusted similarly, with bags instead of labels.

5.4.5. Beyond Walking

We now consider other unrestricted networks (besides walking). In particular, our experiments include a bicycle rental scheme, which can be seen as a hybrid network: It does not have a fixed schedule (and is thus unrestricted), but bicycles can only be picked up and dropped off at designated *cycling stations*. Picking a bike from its station counts as a trip. To handle cycling within MCR, we consider it during the first stage of each round (together with RAPTOR and before walking). Because bicycles have no schedule, we process them independently (from RAPTOR) by running MLC on the bicycle network. To do so, we initialize MLC with labels from round $i-1$ for all relevant bicycle stations and, during the algorithm, we update labels of (the current) round i.

We consider a taxi ride to be a trip as well, since we board a vehicle. Moreover, we also optimize a separate criterion reflecting the (monetary) *cost* of taxi rides. If taxis were not penalized in any way, an all-taxi journey would almost always dominate all other alternatives (even sensible ones), since it is fast and has no walking. Our round-based algorithms handle taxis as they do walking, except that in the taxi stage labels are read from round $i-1$ and written into round i. Note that we link the taxi network to public transit stops as well as bicycle stations and that—unlike with rental bicycles—we also allow taking a taxi as the first and/or last leg of any location-to-location query. Dealing with personal cars or bicycles is simpler. Assuming that they are only available for the first or last legs of the journey, we must only consider them during initialization. Initialization can also handle other special cases, such as allowing rented bicycles to be ridden to the destination (to be returned later).

Note that contraction can be used for cycling and driving. For every unrestricted network (walking, cycling, driving), we keep the link vertices (stops and bicycle stations) in one common core and contract (up to) all other nodes. As before, queries start with upward searches in each relevant unrestricted network.

5.4.6. Heuristics

Even with all accelerations, the exact algorithms proposed in Section 5.4.3 are not fast enough for interactive applications. This section proposes quicker heuristics aimed at finding a set of journeys that is similar to the exact solution, which we take as *ground truth*. We consider three approaches: Weakening the dominance rules, restricting the amount of walking, and reducing the number of criteria. We also discuss how to measure the quality of the heuristic solutions we find.

Weak Dominance. The first strategy we consider is to weaken the domination rules during the algorithm, reducing the number of labels pushed through the network.

We test four implementations of this strategy. The first, MCR-hf, uses fuzzy dominance (instead of strict dominance) when comparing labels during the algorithm: For labels L_1 and L_2, we compute the fuzzy dominance value $d(L_1, L_2)$ (cf. Section 5.4.2) and dominate L_2 if d exceeds a given threshold (we use 0.9). The second, MCR-hb(κ), uses strict dominance, but discretizes criterion κ: Before comparing labels L_1 and L_2, we first round $\kappa(L_1)$ and $\kappa(L_2)$ to predefined discrete values (*buckets*); this can be extended to use buckets for several criteria. The third heuristic, MCR-hs(κ), uses strict dominance but adds a slack of x units to κ. More precisely, L_1 already dominates L_2 if $\kappa(L_1) \leq \kappa(L_2) + x$ and L_1 is at least as good L_2 in all other criteria. The last heuristic, MCR-ht, weakens the domination rule by trading off two or more criteria. More concretely, consider the case in which walking (κ_{walk}) and arrival time (κ_{arr}) are criteria. Then, L_1 already dominates L_2 if $\kappa_{\text{arr}}(L_1) \leq \kappa_{\text{arr}}(L_2) + a \cdot (\kappa_{\text{walk}}(L_1) - \kappa_{\text{walk}}(L_2))$, $\kappa_{\text{walk}}(L_1) \leq \kappa_{\text{walk}}(L_2) + a \cdot (\kappa_{\text{arr}}(L_1) - \kappa_{\text{arr}}(L_2))$, and L_1 is at least as good as L_2 in all other criteria, for a tradeoff parameter a (we use $a = 0.3$).

Restricting Walking. Consider our simple scenario of walking and public transit. Intuitively, most journeys start with a walk to a nearby stop, followed by one or more trips (with short transfers) within the public transit system, and finally a short walk from the final stop to the actual destination. This motivates a second class of heuristics, MCR-tx. It still runs three-criterion search (walking, arrival, and trips), but limits walking transfers between stops to x minutes; in this case we precompute these transfers. MCR-tx-ry also limits walking in the beginning and end of the journey (around s and t) to y minutes. Note that existing solutions often use such restrictions [BCE+10].

Fewer Criteria. The last strategy we study is reducing the number of criteria considered during the algorithm. As already mentioned, this is a common approach in practice. We propose MR-x, which still works in rounds, but optimizes only the number of trips and arrival times explicitly (as criteria). To account for walking duration, we count every x minutes of a walking segment (transfer) as a trip; the first x minutes are free. With this approach, we can run plain Dijkstra to compute transfers, since link vertices no longer need to keep bags. The round index to which labels are written then depends on the walking duration (of the current segment) of the considered label.

A special case is $x = \infty$, where a transfer is never a trip. Another variant is to always count a transfer as a single trip, regardless of duration; we abuse notation and call this variant MR-0. We also consider MR-∞-tx: Walking duration is not an explicit criterion and transfers do not count as trips, but are limited to x minutes.

For scenarios that include cost as a criterion (for taxis), we consider variants of the MCR-hb and MCR-hf heuristics. In both cases, we drop walking as an independent criterion, leaving only arrival time, number of trips, and costs to optimize. We account for walking by making it a (cheap) component of the costs.

5.4.7. Evaluating Quality

To measure the quality of a heuristic from Section 5.4.6, we compare the set of journeys it produces to the *ground truth*, which we define as the solution found by MCR. To do so, we first compute the score of each journey with respect to the Pareto set that contains it (cf. Section 5.4.2). Then, for a given parameter k, we measure the similarity between the top k scored journeys returned by the heuristics and the top k scored journeys in the ground truth. Note that the score depends only on the algorithm itself and does not assume knowledge of the ground truth, which is consistent with a real-world deployment.

To compare two sets of k journeys, we run a greedy maximum matching algorithm. First, we compute a $k \times k$ matrix where entry (i, j) represents the similarity between the i-th journey in the first set and the j-th in the second. To measure the similarity, we make use of the same fuzzy relational operators we use for scoring. More precisely,

given two journeys J_1 and J_2, the similarity with respect to the i-th criterion is given by $c^i := \mu^i_=(\kappa^i(J_1) - \kappa^i(J_2))$, where κ^i is the value of this criterion and $\mu^i_=$ is the corresponding fuzzy equality relation. Then, we define the similarity between J_1 and J_2 as $T(c^1, c^2, \ldots, c^C)$, where T is an arbitrary t-norm (cf. Section 5.4.2). We always select T to be consistent with the s-norm that we use to compute the score values.

After computing the pairwise similarities, we greedily select the unmatched pairs with highest similarity (by picking the highest entry in the matrix that does not share a row or column with a previously picked entry). The similarity of the whole matching is the average similarity of its pairs, weighted by the fuzzy score of the reference journey. This means that matching the highest-scored reference journey is more important than matching the k-th one.

5.4.8. Experiments

This section presents an extensive evaluation of the methods introduced in the previous subsections. All algorithms from Sections 5.4.3 and 5.4.6 were implemented in C++ and compiled with g++ 4.6.2 (64 bits, flag `-O3`). We ran our experiments on one core of a dual 8-core Intel Xeon E5-2670 clocked at 2.6 GHz, with 64 GiB of DDR3-1600 RAM.

Inputs. We focus on the transportation network of London (England); results for other instances are similar. We use the same timetable instance as in Section 4.6.8. The data was made available by Transport for London (TfL) [Lon11, Tra00], from which we extracted a Tuesday in the periodic summer schedule of 2011. The data includes subway (tube), buses, tram, ferries, and light rail (DLR), as well as bicycle station locations. To model the underlying road network, we use data provided by PTV AG [PTV79] from 2006, which explicitly indicates whether each road segment is open for driving, cycling and/or walking. We set the walking speed to 5 km/h and the cycling speed to 12 km/h, and we assume driving at free-flow speeds. We do not consider turn costs, which are not defined in the data. The resulting combined network has 564 cycle stations and about 20 k stops, 5 M departure events, and 259 k vertices in the walking network. Exact numbers are given in Table 5.7.

Parameterizing Fuzziness. Recall that we specify the fuzziness of each criterion by a pair (χ, ϵ), roughly meaning that the corresponding Gaussian (centered at $x = 0$) has value χ for $x = \epsilon$. We set these pairs to $(0.8, 5)$ for walking, $(0.8, 1)$ for arrival time, $(0.1, 1)$ for trips, and $(0.8, 5)$ for costs (given in pounds; times are in minutes). Note that the number of trips is sharper than the other criteria. Later in this section we show that our approach is robust to small variations in these parameters, but they can be tuned to account for user-dependent preferences. If not indicated otherwise, our experiments consider the minimum/maximum norms by default.

Table 5.7. Size figures for our input instances. We link every stop and cycle station with the walking/taxi network.

Figure	London	New York	Los Angeles	Chicago
PUBLIC TRANSIT:				
Stops	20 843	17 894	15 003	12 137
Routes	2 184	1 393	1 099	710
Trips	133 011	45 299	16 376	20 303
Daily Departure Events	4 991 125	1 825 129	931 846	1 194 571
Vertices (Route Model)	99 230	66 124	81 657	47 561
Edges (Route Model)	260 583	193 159	214 369	118 452
WALKING:				
Vertices	258 840	255 808	224 053	70 440
Vertices in Core	27 840	25 808	21 053	16 440
Edges	1 433 814	1 586 782	1 395 185	586 979
Footpaths $\leq$ 5 min	150 948	219 040	83 844	122 450
Footpaths $\leq$ 10 min	518 174	670 702	271 444	426 818
CYCLING:				
Cycle Stations	564	—	—	—
Vertices	23 311	—	—	—
Vertices in Core	1 311	—	—	—
Edges	130 971	—	—	—
TAXI:				
Vertices	259 122	—	—	—
Vertices in Core	27 122	—	—	—
Edges	1 339 487	—	—	—

Methodology. We ran *location-to-location* queries, with sources, targets, and departure times picked uniformly at random (from the walking network and during the day, respectively).

Algorithms Evaluation

For our first experiment, we use walking, cycling, and the public transportation network and consider three criteria: arrival time, number of trips, and walking duration. We ran 1 000 queries for each algorithm. Table 5.8 summarizes the results (additional statistics are discussed later). For each algorithm, the table first shows which criteria are explicitly taken into account. The next five columns show the average values observed for the number of rounds, scans per entity (stop/vertex), label comparisons per entity, journeys found, and running time (in milliseconds). The last four columns

Table 5.8. Performance and solution quality on journeys considering walking, cycling, and public transit. Bullets (•) indicate the criteria taken into account by the algorithm.

Algorithm	Arr.	Trp.	Wlk.	Rnd.	Scans / Ent.	Comp. / Ent.	Jn.	Time [ms]	Quality-3 Avg.	Quality-3 Sd.	Quality-6 Avg.	Quality-6 Sd.
MCR-full	•	•	•	13.8	13.8	168.2	29.1	4 634.0	100 %	0 %	100 %	0 %
MCR	•	•	•	13.8	3.4	158.7	29.1	1 438.7	100 %	0 %	100 %	0 %
MLC	•	•	•	—	10.6	1 246.7	29.1	4 543.0	100 %	0 %	100 %	0 %
MCR-hf	•	•	•	15.6	2.9	14.3	10.9	699.4	89 %	15 %	89 %	11 %
MCR-hb	•	•	•	10.2	2.1	12.7	9.0	456.7	91 %	12 %	91 %	10 %
MCR-hs	•	•	•	14.7	2.6	11.1	8.6	466.1	67 %	28 %	69 %	23 %
MCR-ht	•	•	•	10.5	2.0	6.4	8.6	373.6	84 %	22 %	82 %	20 %
MCR-t10	•	•	•	13.8	2.7	132.7	29.0	1 467.6	97 %	10 %	95 %	10 %
MCR-t10-r15	•	•	•	10.7	1.7	73.3	13.2	885.0	38 %	40 %	30 %	31 %
MCR-t5	•	•	•	13.8	2.7	126.6	28.9	891.9	93 %	16 %	92 %	15 %
MR-∞	•	•	○	7.6	1.4	4.8	4.5	44.4	63 %	28 %	63 %	24 %
MR-0	•	•	○	13.7	2.1	6.9	5.4	61.5	63 %	28 %	63 %	24 %
MR-10	•	•	○	20.0	1.1	4.8	4.3	39.4	51 %	33 %	45 %	29 %
MR-∞-t10	•	•	○	7.6	1.1	4.8	4.5	22.2	63 %	28 %	62 %	24 %

evaluate the quality of the top 3 and 6 journeys found by our heuristics, as explained in Section 5.4.6. Note that we show both averages and standard deviations.

The methods in Table 5.8 are grouped in blocks. Those in the first block compute the full Pareto set considering all three criteria (arrival time, number of trips, and walking). MCR, our reference algorithm, is round-based and uses contraction in the unrestricted networks. As anticipated, it is faster (by a factor of about three) than MCR-full (which does not use the core) and MLC (which uses the core but is not round-based). Accordingly, all heuristics we test are round-based and use the core.

The second block contains heuristics that accelerate MCR by weakening the domination rules, causing more labels to be pruned (and losing optimality guarantees). As explained in Section 5.4.6, MCR-hf uses fuzzy dominance during the algorithm, MCR-hb uses walking *buckets* (discretizing walking by steps of 5 minutes for domination), MCR-hs uses a slack of 5 minutes on the walking criterion when evaluating domination, and MCR-ht considers a tradeoff parameter of $a = 0.3$ between walking and arrival time. All heuristics are faster than pure MCR, and MCR-hb gives the best quality at a reasonable running time.

The third block has algorithms with restrictions on walking duration. Limiting transfers to 10 minutes (as MCR-t10 does) has almost no effect on solution quality (which is expected in a well-designed public transportation network). Moreover, adding precomputed footpaths of 10 minutes is not faster than using the core for unlimited walking (as MCR does).

Table 5.9. Detailed performance analysis of our algorithms. The total running time includes additional overhead, such as for initialization.

				Public Transit		Walking		Cycling		Total	
Algorithm	Arr.	Trp.	Wlk.	Scans / Stop	Time [ms]	Scans / Vert.	Time [ms]	Scans / Vert.	Time [ms]	Scans / Ent.	Time [ms]
MCR-full	•	•	•	32.1	350.6	9.6	3 030.9	43.6	1 203.1	13.8	4 634.0
MCR	•	•	•	32.1	341.4	1.2	889.3	1.7	159.2	3.4	1 438.7
MLC	•	•	•	119.3	—	2.6	—	2.1	—	10.6	4 543.0
MCR-hf	•	•	•	28.1	157.7	1.0	483.9	0.7	25.6	2.9	699.4
MCR-hb	•	•	•	21.1	115.2	0.7	297.4	0.5	19.7	2.1	456.7
MCR-hs	•	•	•	25.1	97.3	0.9	322.2	0.6	16.8	2.6	466.1
MCR-ht	•	•	•	20.2	86.8	0.7	246.4	0.5	17.4	2.0	373.6
MCR-t5	•	•	•	31.5	318.4	0.5	348.6	1.7	157.2	2.7	891.9
MCR-t10	•	•	•	31.6	326.2	0.5	913.7	1.7	158.5	2.7	1 467.6
MCR-t10-r15	•	•	•	20.0	207.5	0.3	554.0	1.2	103.6	1.7	885.0
MR-∞	•	•	○	14.2	10.0	0.5	31.0	0.3	1.8	1.4	44.4
MR-0	•	•	○	21.4	13.9	0.7	42.5	0.4	2.4	2.1	61.5
MR-10	•	•	○	9.7	6.3	0.5	30.5	0.2	1.3	1.1	39.4
MR-∞-t10	•	•	○	14.4	9.4	0.2	9.5	0.3	1.6	1.2	22.2

Additionally limiting the walking range from s or t (MCR-t10-r15) improves speed, but the quality becomes unacceptably low: The algorithm misses good journeys (including all-walk) quite often. If instead we allow even more restricted transfers (with MCR-t5), we get a similar speedup with much better quality (comparable to MCR-hb).

The MR-x algorithms (fourth block) reduce the number of criteria considered by combining trips and walking. The fastest variant is MR-∞-t10, which drops walking duration as a criterion but limits the amount of walking at transfers to 10 minutes, making it essentially the same as RAPTOR (cf Section 4.6), with a different initialization. As expected, however, quality is much lower than for MCR-tx, confirming that considering the walking duration explicitly during the algorithm is important to obtain a full range of solutions. MR-10 attempts to improve quality by transforming long walks into extra trips, but is not particularly successful.

Summing up, MCR-hb should be the preferred choice for high-quality solutions, while MR-∞-t10 can support interactive queries with reasonable quality.

Detailed Performance

Table 5.9 presents a more detailed analysis of the previous experiment (still without taxis). For each algorithm, it shows the effort (number of scans per vertex and/or stop, as well as running times in milliseconds) spent in each of the networks (public

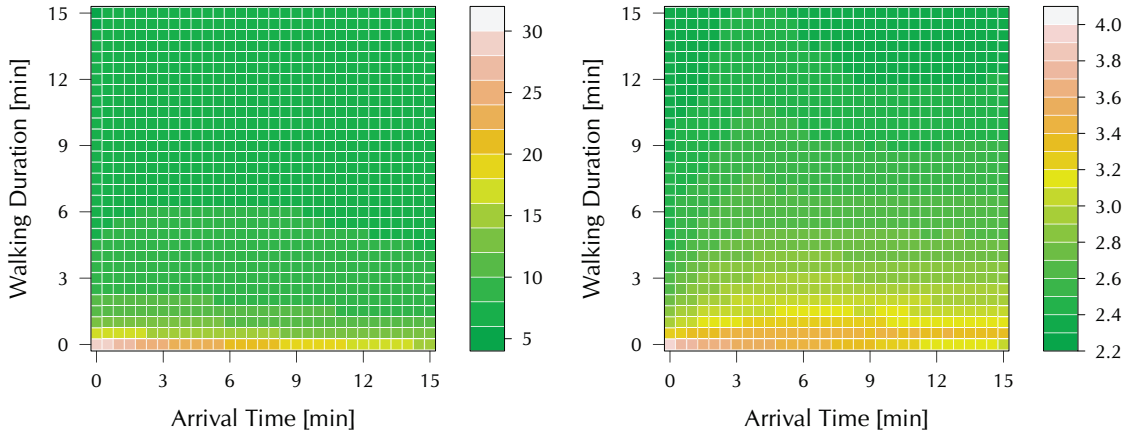


Figure 5.14. Number of Pareto-optimal journeys with score higher than 0.1 for varying fuzziness. We consider both the maximum norm (left) and probabilistic sum (right). The x axis varies the fuzziness in the arrival time, while the y axis considers the walking duration. The intensity (color) of the corresponding entry indicates the average number of journeys in the filtered output.

transit, walking, and cycling) and in total. The table shows that all round-based algorithms except MR-∞-t10 spend significantly more time processing the unrestricted networks (walking and cycling) than dealing with public transportation. This was to be expected: not only are the unrestricted networks bigger (they have more vertices), but also they must be processed with a (slower) Dijkstra-based algorithm (as in MLC, rather than RAPTOR). This is the reason for the good performance of the MR-∞-t10 heuristic.

Fuzzy Parameters Evaluation

We also evaluated the impact of the fuzzy parameters on the number of journeys we obtain. We again use London with walking, public transit, and cycling as input. Figure 5.14 shows the number of journeys given a score higher than 0.1 (by the fuzzy ranking routine) when we vary ϵ (the level of fuzziness) for two criteria, walking and arrival time. We set $\chi = 0.8$, as in our main experiments. To not overload the figure, we keep the fuzziness of the third criterion (number of trips) constant.

A comparison between the plots shows that, for the same set of parameters, probabilistic sum is significantly stricter than the maximum norm and reduces the number of journeys much more drastically (for a fixed threshold). Qualitatively, however, they behave similarly. Under both norms, making the walking criterion fuzzier is more effective at identifying unwanted journeys. A couple of minutes of fuzziness in the walking criterion is enough to significantly reduce the number of

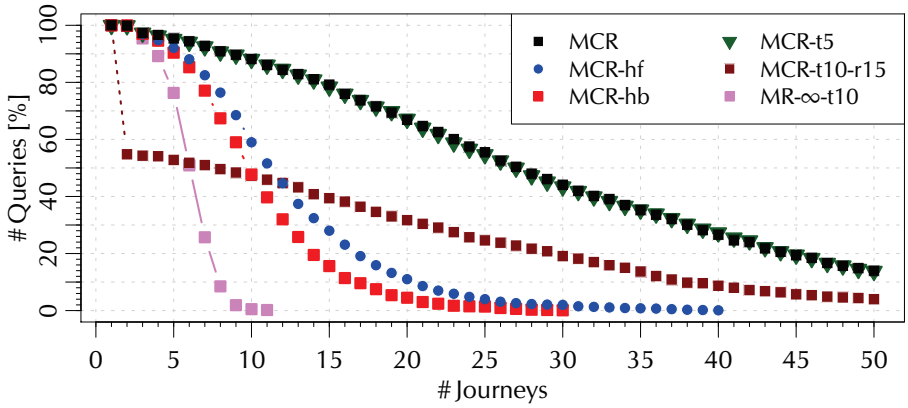


Figure 5.15. Evaluating the number of journeys returned by some of our algorithms: For a given n (on the abscissa), we report the percentage of 1 000 random queries that compute n or more journeys.

journeys above the threshold. Adding fuzziness only to the arrival time has much more limited effect on the results.

Quality of the Heuristics

We here further investigate the quality of our heuristics. We use London with walking, public transit, and cycling as input. Figure 5.15 reports the size of the Pareto set (the input to scoring) for various algorithms, while Figure 5.16 shows how well the the top k heuristic journeys match the ground truth, for varying k. We observe that exact MCR (even if restricted to 5-minute transfers) does indeed produce many journeys, supporting the notion of ranking them afterwards (by score). A good heuristic, such as MCR-hb, computes much fewer journeys, but they match the top MCR journeys quite well. An interesting observation is that the quality of the heuristic hardly depends on the number of journeys we try to match.

Full Multimodal Problem

Our next experiment considers the full multimodal problem, including taxis. We add *cost* as fourth criterion (at 2.40 pounds per taxi trip plus 60 pence per minute). We do not consider the cost of public transit, since it is significantly cheaper. Table 5.10 presents the average performance of some of our algorithms over 1 000 random queries in London. The first block includes algorithms that optimize all four criteria (arrival time, walking duration, number of trips, and costs). While exact MCR is impractical, fuzzy domination (MCR-hf) makes the problem tractable with little loss in quality. Using 5-minute buckets for walking and 5-pound buckets for costs (MCR-hb) is even faster, though queries still take more than two seconds.

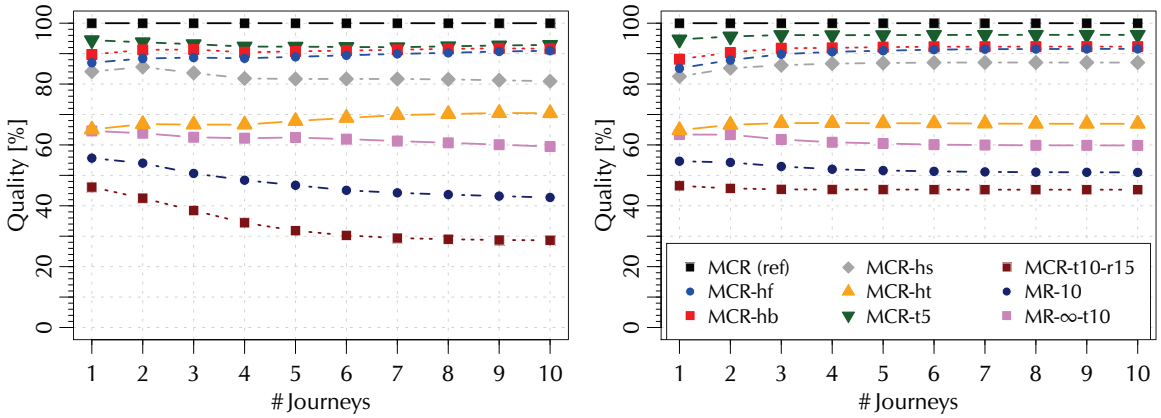


Figure 5.16. Evaluating the solution quality by matching the top k journeys in the solution with the top k of the reference algorithm (MCR). The scores and similarity values are obtained by using the minimum/maximum norms (left) and the product norm/probabilistic sum (right). The key of the right plot also applies to the left.

The second block shows that we can reduce running times by dropping walking duration as a criterion (we incorporate it into the cost function at 3 pence per minute, instead), with almost no loss in solution quality. This is still not fast enough, though. Using 5-pound buckets (MCR-hb) reduces the average query time to about 1 second, with reasonable quality.

Additional Inputs

In addition to London, we tested inputs representing other large metropolitan areas (New York, Los Angeles, and Chicago). We built the public transit network from publicly available General Transit Feeds (GTFS) [Gen10], restricting ourselves to the timetable for August 10, 2011 (a Wednesday). The walking network data is still given by PTV [PTV79], and these instances do not include bicycles. Detailed statistics for

Table 5.10. Performance on our London instance when taking taxi into account.

Algorithm	Arr.	Trp.	Wlk.	Cost	Rnd.	Scans /Ent.	Comp. /Ent.	Jn.	Time [ms]	Quality-3 Avg.	Quality-3 Sd.	Quality-6 Avg.	Quality-6 Sd.
MCR	●	●	●	●	16.3	3.1	369 606.0	1 666.0	1 960 234.0	100 %	0 %	100 %	0 %
MCR-hf	●	●	●	●	17.1	2.1	137.1	35.2	6 451.6	92 %	12 %	92 %	6 %
MCR-hb	●	●	●	●	9.9	1.3	86.8	27.6	2 807.7	96 %	8 %	92 %	6 %
MCR	●	●	○	●	14.6	2.4	7 901.4	250.9	25 945.8	98 %	6 %	97 %	5 %
MCR-hf	●	●	○	●	12.0	1.4	33.6	17.6	2 246.3	87 %	12 %	74 %	12 %
MCR-hb	●	●	○	●	9.0	1.0	20.0	11.6	996.4	86 %	12 %	74 %	12 %

all instances are presented in Table 5.7 at the beginning of this section.

Table 5.11 compares the performance of our algorithms on these inputs. For reference, we also consider a simplified version of the London network, without bicycles. For each input, we show the average values (over 1 000 queries) for number of journeys found, running time, and quality (considering the top 6 journeys). The results are consistent with those obtained for the full London network, showing that our preferred choice of heuristics also holds here. MCR-hb is always the best choice in terms of solution quality (among methods with reasonable speedups), while MR-∞-t10 is preferred if query times should be as low as possible.

5.4.9. Conclusion

In this section, we have studied multicriteria journey planning in multimodal networks. We argued that users optimize two classes of criteria: arrival time, and mode-dependent convenience. Although the corresponding full Pareto set is large and has many unnatural journeys, fuzzy set theory can extract the relevant journeys and rank them. Since exact algorithms are too slow, we have introduced several heuristics that closely match the best journeys in the Pareto set. Our experiments show that our approach enables efficient realistic multimodal journey planning in large metropolitan areas.

A natural avenue for future research is accelerating our approach further to enable interactive queries with an even richer set of criteria in dynamic scenarios, handling delay and traffic information. The ultimate goal would be to compute multicriteria multimodal journeys on a global scale in real time.

Besides accelerating the algorithm, we are also interested in objective criteria that evaluate the significance of a journey. (Note we compute ranks with respect to the other journeys of the Pareto set.) First ideas of objectively classifying journeys are presented in [BBS13].

Table 5.11. Evaluating the performance of MCR and MR with different heuristics on other instances. The quality is determined identically to Table 5.8.

				London No Bike			New York			Los Angeles			Chicago		
Algorithm	Arr.	Trp.	Wlk.	Jn.	Time [ms]	Qual. Avg.	Jn.	Time [ms]	Qual. Avg.	Jn.	Time [ms]	Qual. Avg.	Jn.	Time [ms]	Qual. Avg.
MCR	●	●	●	27.5	1 215.9	100 %	25.5	1 703.0	100 %	16.7	644.6	100 %	22.1	532.8	100 %
MCR-hf	●	●	●	10.5	677.3	89 %	8.6	611.0	91 %	8.9	445.0	88 %	8.3	241.3	72 %
MCR-hb	●	●	●	8.7	430.3	91 %	7.2	413.8	94 %	7.6	295.8	93 %	7.1	160.8	92 %
MCR-hs	●	●	●	8.5	450.6	68 %	6.7	414.0	84 %	7.4	310.7	62 %	6.6	158.8	58 %
MCR-ht	●	●	●	8.3	342.6	81 %	6.6	300.9	80 %	6.7	228.4	69 %	6.2	113.9	79 %
MCR-t5	●	●	●	27.3	671.7	94 %	25.6	695.5	69 %	16.6	262.7	93 %	21.9	277.7	95 %
MCR-t10	●	●	●	27.4	1 123.0	96 %	25.3	1 401.4	85 %	16.8	424.5	96 %	22.0	578.8	98 %
MCR-t10-r15	●	●	●	11.9	688.1	28 %	5.4	677.9	10 %	3.9	202.0	13 %	9.6	372.7	28 %
MR-∞	●	●	○	4.4	40.0	61 %	3.4	26.3	65 %	3.6	21.5	51 %	3.3	12.3	63 %
MR-0	●	●	○	5.2	55.7	61 %	3.8	37.6	65 %	4.3	28.5	52 %	3.7	15.6	63 %
MR-10	●	●	○	6.1	36.8	43 %	6.0	26.1	41 %	6.1	26.6	42 %	5.1	13.9	50 %
MR-∞-t10	●	●	○	4.4	19.7	61 %	3.6	10.6	60 %	3.6	11.0	51 %	3.3	7.1	63 %

Chapter 6

Customizable Route Planning in Road Networks

The past decade has seen a great deal of research on finding point-to-point shortest paths on road networks (cf. Section 2.1). Although Dijkstra's algorithm [Dij59] runs in almost linear time with very little overhead, it still takes a few seconds on continental-sized graphs. Practical algorithms use a two-stage approach: *preprocessing* takes a few minutes (or even hours) and produces a (linear) amount of auxiliary data, which is then used to perform *queries* in real time. Most previous research focused on the most natural metric, driving times. Real-world systems, however, often support other natural metrics as well, such as shortest distance, walking, biking, avoid U-turns, avoid/prefer freeways, or avoid left turns.

In this chapter, we consider the *customizable route planning* problem, whose goal is to perform real-time queries on road networks with *arbitrary metrics*. Such algorithms can be used in two scenarios: They may keep several active metrics at once (to answer queries for any of them), or new metrics can be generated on the fly. A system with these properties has obvious attractions. It supports real-time traffic updates and other dynamic scenarios, allows easy customization by handling any combination of standard metrics, and can even provide personalized driving directions (for example, for a truck with height and weight restrictions). To implement such a system, we need an algorithm that allows real-time queries, has fast customization (a few seconds), and keeps very little data for each metric. Most importantly, it must be *robust*: All three properties must hold for *any metric*. No existing algorithm meets these requirements.

Contributions. To achieve the aforementioned goals, we distinguish between two features of road networks. The *topology* is a set of static properties of each road segment or turn, such as physical length, road category, speed limits, and turn types. The *metric* encodes the actual cost of traversing a road segment or taking a turn. It

can often be described compactly, as a function that maps (in constant time) the properties of an arc/turn into a cost. We assume the topology is shared by the metrics and rarely changes, while metrics may change quite often and even coexist.

To exploit this separation, we consider algorithms for customizable route planning with *three stages*. The first, *metric-independent preprocessing*, may be relatively slow, since it is run infrequently. It takes only the graph topology as input and may produce a fair amount of auxiliary data (comparable to the input size). The second stage, *metric customization*, is run once for each metric, must be much quicker (a few seconds), and produce little data—a small fraction of the original graph. Finally, the *query stage* uses the outputs of the first two stages and must be fast enough for real-time applications.

In Section 6.1 we explore the design space by analyzing the applicability of existing algorithms to this setting. We note that methods with a strong hierarchical component, the fastest in many situations, are too sensitive to metric changes. We focus on separator-based methods, which are more robust but have often been neglected in recent research, since published results made them seem uncompetitive: The highest speedups over Dijkstra observed were lower than 60 [HSW08], compared to thousands or millions with other methods.

Section 6.2 revisits and thoroughly reengineers a separator-based algorithm. By applying existing acceleration techniques, recent advances in graph partitioning—and some engineering effort—we can answer queries on continental road networks in about a millisecond, with much less customization time (a few seconds) and space (a few tens of megabytes) than existing acceleration techniques.

Another contribution of this chapter is a careful treatment of turn costs (Section 6.4). It has been widely believed that any algorithm can be easily augmented to handle these efficiently, but we note that some methods actually have a significant performance penalty, especially if turns are represented space-efficiently. In contrast, we can handle turns naturally, with little effect on performance.

We stress that our algorithms are not meant to be the fastest on any particular metric. For "nice" metrics, our queries are slower than the best hierarchical methods. However, our queries are robust and suitable for real-time applications with arbitrary metrics, including those for which the hierarchical methods fail. Our method can quickly process new metrics, and the metric-specific information is small enough to keep several metrics in memory at once.

References. The author of this thesis worked on this chapter while visiting Microsoft Research Silicon Valley. He contributed to Sections 6.1 and 6.2. Sections 6.3 and 6.4 have been developed without the author by the time he left Microsoft Research. The content of this chapter is based on [DGPW11], which appeared at the 10th International Symposium on Experimental Algorithms (SEA'11). A journal version [DGPW14] has been accepted for publication in the INFORMS Journal for

Transportation Science. This chapter is joint work with Daniel Delling, Andrew Goldberg, and Renato Werneck.

We also like to thank Ittai Abraham and Ilya Razenshteyn for their valuable input, and Christian Vetter for sharing his Contraction Hierarchies results with us.

6.1. Analysis of Previous Algorithms

There has been previous work on variants of the route planning problem that deal with multiple metrics in a nontrivial way. The preprocessing of SHARC [BD09] can be modified to handle multiple (known) metrics at once. In the *flexible routing problem* [GKS10], one must answer queries on linear combinations of a small set of metrics (typically two) known in advance. Queries in the *constrained routing problem* [RT10] must avoid entire classes of arcs. In multicriteria optimization [DW09a], one must find Pareto-optimal paths among multiple metrics. ALT [GH05] and CH [GSSV12] can adapt to small changes in a benign base metric without rerunning preprocessing in full. All these approaches must know the base metrics in advance, and for good performance the metrics must be few, well-behaved, and similar to one another. In practice, even seemingly small changes to the metric (such as higher U-turn costs) render some approaches impractical. In contrast, we must process metrics as they come and assume nothing about them.

We now discuss the properties of existing point-to-point algorithms to determine how well they fit our design goals. Some of the most successful existing methods—such as reach-based routing [GKW09], Contraction Hierarchies (CH) [GSSD08], SHARC [BD09], Transit Node Routing [BFM+07], and Hub Labels [ADGW11]—rely on the strong *hierarchy* of road networks with travel times: The fastest paths between faraway regions of the graph tend to use the same major roads.

For metrics with strong hierarchies, such as travel times, CH has many of the features we want. During preprocessing, CH heuristically sorts the vertices in increasing order of importance and *shortcuts* them in this order. (To *shortcut* a vertex v, we temporarily remove it from the graph and add arcs as necessary to preserve the distances between its neighbors.) Queries run bidirectional Dijkstra, but only follow arcs or shortcuts to more important vertices. If a metric changes only slightly, one can keep the order and recompute the shortcuts in about a minute [GSSD08]. Unfortunately, an order that works for one metric may not work for a substantially different one (e. g., travel times and distances, or a major traffic jam). Furthermore, queries are much slower on metrics with less-pronounced hierarchies [BDS+10]. More crucially, the preprocessing stage can become impractical (in terms of space and time) for bad metrics, as Section 6.4 will show.

In contrast, techniques based on *goal direction*, such as PCD [MSM09], ALT [GH05], and Arc Flags [HKMS09], produce the same amount of auxiliary data for any metric. Queries are not robust, however: They can be as slow as Dijkstra for bad metrics.

Table 6.1. Rough categorization of existing algorithms. In the columns *one good metric*, we report the estimated relative performance of the algorithms when optimizing a single well-behaved metric, such as travel times, starting from scratch. We look at the size of the auxiliary data (C.-Space), customization times (C.-Time), and query times (Q.-Time). In *arbitrary metrics*, we consider the same values when dealing with several arbitrarily bad metrics. Also compare to Figure 2.12

	One Good Metric			Arbitrary Metrics		
Algorithm	C.-Space	C.-Time	Q.-Time	C.-Space	C.-Time	Q.-Time
Dijkstra [Dij59]	+++	+++	−−−	+++	+++	−−−
PCD [MSM09]	+++	++	−−	+++	++	−−
ALT [GH05]	○	++	−	○	++	−−
Reach [Gut04]	++	+	+	−−−	−−−	−−−
Arc Flags [Lau04]	+	++	+	○	−−	−
SHARC [BD09]	++	++	++	○	−−	−
CH [GSSV12]	+++	+++	++	−−−	−−−	−−−
HPML [DHM+09]	−−	−−−	+++	−	−	++
TNR [SS09]	−	○	+++	−−−	−−−	−−
Hub Labels [ADGW11]	−−	−−	+++	−−−	−−−	−−−
MLD (this chapter)	+++	+++	+	+++	+++	+

Even for travel times, PCD and ALT are not competitive with other methods.

A third approach is based on *graph separators* [HJR96, JP02, SWZ02, HSW08]. During preprocessing, one computes a multilevel partition of the graph to create a series of interconnected overlay graphs. A query starts at the lowest (local) level and moves to higher (global) levels as it progresses. These techniques predate hierarchy-based methods, but their query times are widely regarded as uncompetitive in practice, and they have not been tested on continental road networks. (The exceptions are recent extended variants [DHM+09, MZ07] that achieve great query times by adding many more arcs during preprocessing, which is costly in time and space.) Because preprocessing and query times are essentially metric-independent, separator-based methods are the most natural fit for our problem.

Finally, Table 6.1 gives a very rough summary of previous algorithms (cf. Section 2.1) and our proposed approach (MLD) regarding metric-dependent customization space, customization time, and query time. None of the previous algorithms give a practical solution for our scenario.

6.2. Our Approach to Customizable Route Planning

We will first describe a basic algorithm, then consider several techniques to make it more practical, using experimental results to guide our design. Our code is written in C++ (with OpenMP for parallelization) and compiled with Microsoft

Visual C++ 2010. We use 4-heaps as priority queues. Experiments were run on a commodity workstation with an Intel Core-i7 920 (four cores clocked at 2.67 GHz and 6 GiB of DDR3-1066 RAM) running Windows Server 2008 R2. Our standard benchmark instance is the European road network, with 18 million vertices and 42 million arcs, made available by PTV AG [PTV79] for the 9th DIMACS Implementation Challenge [DGJ09]. Vertex IDs and arc costs are both 32-bit integers.

We must minimize *metric customization time*, *metric-dependent space* (excluding the original graph), and *query time*, while keep metric-independent time and space reasonable. We evaluate our algorithms on 10 000 s–t queries with s and t picked uniformly at random. We focus on finding shortest path *costs*; Section 6.4 shows how to retrieve the actual paths. We report results for travel times and travel distances, but *by design* our algorithms work well for any metric.

6.2.1. Basic Algorithm

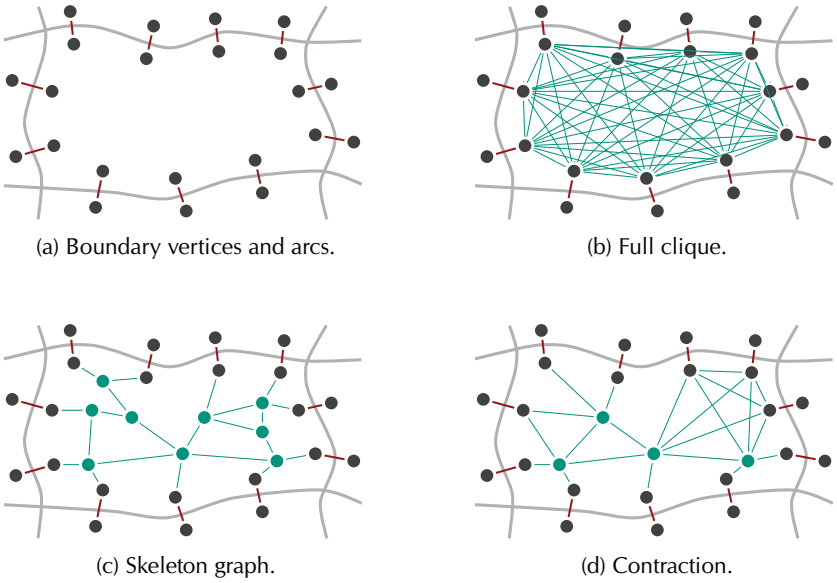
Our *metric-independent preprocessing* stage partitions the graph into connected cells with at most U (an input parameter) vertices each, with as few boundary arcs (arcs with endpoints in different cells) as possible.

Given a partition $\mathcal{C} = (C_1, \ldots, C_k)$, the *metric customization* stage builds a graph H containing all boundary vertices (those with at least one neighbor in another cell) and cut arcs of G. See Figure 6.1a. It also contains a *clique* for each cell C of the partition: for every pair (u, v) of boundary vertices in C, it creates an arc (u, v) whose cost is the same as the shortest path (restricted to C) between u and v (or infinite if v is not reachable from u). See Figure 6.1b. We do so by running Dijkstra's algorithm from each boundary vertex. Note that H is an *overlay* [SWW00]: The distance between any two vertices in H is the same as in G.

Finally, to perform a *query* between s and t, we run a bidirectional version of Dijkstra's algorithm on the graph consisting of the union of H, $C(s)$, and $C(t)$. Here, $C(u)$ denotes the subgraph of G induced by the vertices in the cell containing u. The fact that H is an overlay of G ensures that queries are correct.

As already mentioned, this is the basic strategy of separator-based methods. In particular, HiTi [JP02] uses arc-based separators and cliques to represent each cell. Unfortunately, HiTi has not been tested on large road networks; experiments were limited to small grids, and the original proof of concept does not appear to have been optimized using modern algorithm engineering techniques.

Our first improvement over HiTi and similar algorithms concerns the partition. We use PUNCH [DGRW11]. Recently developed to deal with road networks, it routinely finds solutions with half as many boundary arcs (or fewer), compared to the general-purpose partitioners (such as METIS [KK99] or planar separators [LT79]) commonly used by previous algorithms. Better partitions reduce customization time and space, leading to faster queries. For our experiments, we used relatively long runs of PUNCH, taking about an hour. Our results would not change much if we



(a) Boundary vertices and arcs. (b) Full clique.

(c) Skeleton graph. (d) Contraction.

Figure 6.1. Three possible overlay graphs to represent a cell: *cliques*, *skeleton*, and *contraction*.

used the basic version of PUNCH, which is only about 5 % worse but runs in mere minutes.

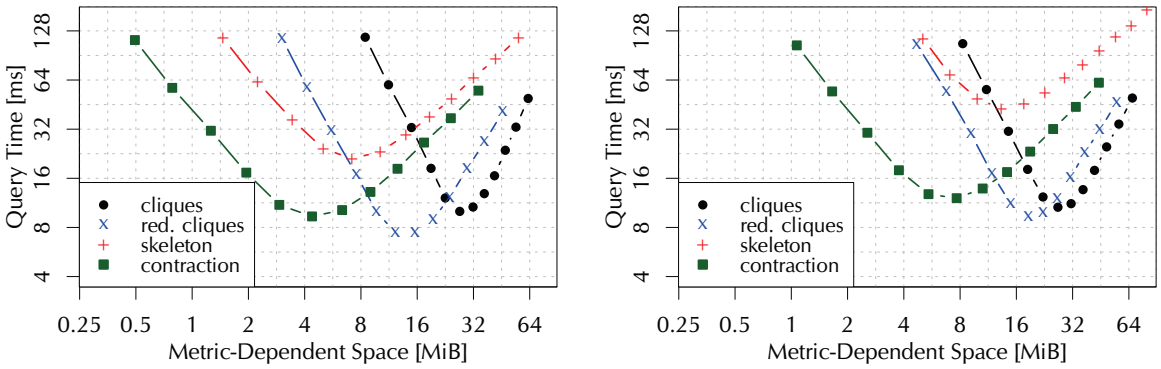
We also use parallelism: Queries run forward and reverse searches on two CPU cores, and customization uses all four cores of our machine (each cell is processed independently).

6.2.2. Overlay Sparsification

Using full cliques in the overlay graph may seem wasteful, particularly for well-behaved metrics. At the cost of making its topology metric-dependent, we consider various techniques to reduce the overlay graph.

Edge Reduction. The first approach is *edge reduction* [SWW00], which eliminates clique arcs that are not shortest paths. After computing all cliques, we run Dijkstra's algorithm from each vertex u in H, stopping as soon as all neighbors of u (in H) are scanned. Note that these searches are usually quick, since they only visit the overlay.

Skeleton Graphs. A more aggressive technique is to preserve some internal cell vertices [DHM+09, HSW08, SWZ02]. If $B = \{u_1, u_2, \ldots, u_k\}$ is the set of boundary vertices of a cell, let T_i be the shortest path tree (restricted to the cell) rooted at u_i, and let T_i' be the subtree of T_i consisting of the vertices with descendants in B. We take the union $C = \cup_{i=1}^{k} T_i'$ of these subtrees and shortcut all internal vertices with



(a) Travel time metric. (b) Distance metric.

Figure 6.2. Effect of sparsification for travel time and distance metric. The i-th data point from the left indicates a partition for $U = 2^{20-i}$.

two neighbors or fewer. Note that this *skeleton graph* is technically not an overlay, but it preserves distances between all *boundary* vertices, which is what we need.

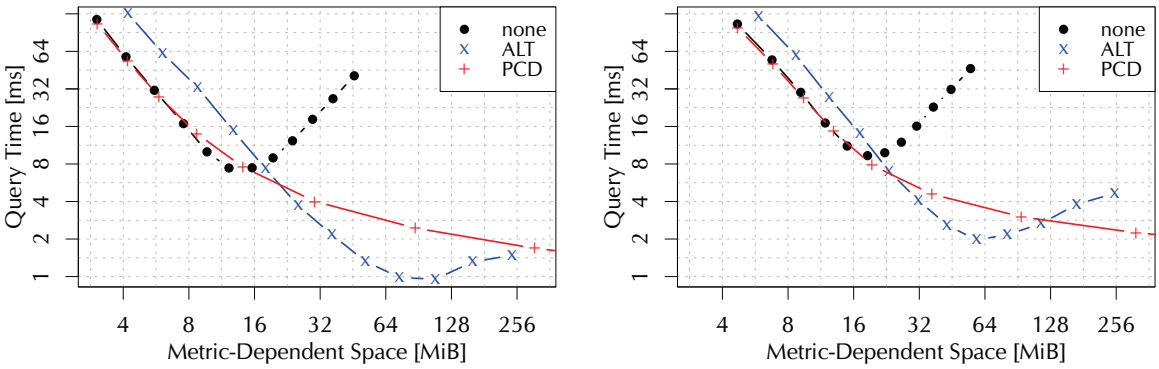
Contraction. Finally, we tried a lightweight *contraction* scheme. Starting from the skeleton graph, we greedily shortcut low-degree internal vertices, stopping when no such operation is possible without increasing the number of arcs by more than one.

Figure 6.1 illustrates the various overlay types.

Evaluation. Figure 6.2 compares all four overlays (cliques, reduced cliques, skeleton, and contraction) on travel times and travel distances. Each plot relates the total query time and the amount of metric-independent data for different values of U (the cell size). Unsurprisingly, all overlays need more space as the number of cells increases (i. e., as U decreases). Query times, however, are minimized when the effort spent on each level is balanced, which happens for $U \approx 2^{15}$.

To analyze preprocessing times (not depicted in the plots), take $U = 2^{15}$ (with travel times) as an example. Finding full cliques takes only 40.8 s, but edge reduction (45.8 s) or building the skeleton graph (45.1 s) are almost as cheap. Contraction, at 79.4 s, is significantly more expensive, but still practical. Most methods get faster as U gets smaller: Full cliques take less than 5 s with $U = 256$. The exception is contraction: When U is very small, the combined size of all skeletons is quite large, and processing them takes minutes.

In terms of query times and metric-dependent space, however, contraction dominates pure skeleton graphs. Decreasing the number of arcs (from 1.2 M with reduced cliques to 0.8 M with skeletons, for $U = 2^{15}$ with travel times) may not be enough to offset an increase in the number of vertices (from 34 k to 280 k), to which Dijkstra-



(a) Travel time metric. (b) Distance metric.

Figure 6.3. Effect of goal-direction for travel time and distance metric. Equal to Figure 6.2, the i-th data point from the left indicates a partition for $U = 2^{20-i}$.

based algorithms are more sensitive. This also explains why reduced cliques yield the fastest queries, with full cliques not far behind.

All overlays have worse performance when we switch from travel times to distances (with less pronounced hierarchies), except full cliques. Since edge reduction is relatively fast, we use reduced cliques as the default overlay.

6.2.3. Goal-Direction

For even faster queries, we can apply more sophisticated techniques (than bidirectional Dijkstra) to search the overlay graph. While in principle any method could be used, our model restricts us to those with metric-independent preprocessing times. We tested Precomputed Cluster Distances (PCD) and ALT (cf. Section 2.1.2).

Precomputed Cluster Distances. To use Precomputed Cluster Distances (PCD), which have been introduced in [MSM09], with our basic algorithm, we do the following. Let k be the number of cells found during the metric independent preprocessing ($k \approx |V|/U$). During metric customization, we run Dijkstra's algorithm k times on the overlay graph to compute a $k \times k$ matrix with the distances between all cells. Queries then use the matrix to guide the bidirectional search by pruning vertices that are far from the shortest path. Note that, unlike "pure" PCD, we use the overlay graph during customization and queries.

Core-ALT. Another technique is *Core-ALT* (CALT) [BDS+10]. Queries start with bidirectional Dijkstra searches restricted to the source and target cells. Their boundary vertices are then used as starting points for an ALT (A* search using landmarks and the triangle inequality) query on the overlay graph. The ALT preprocessing runs

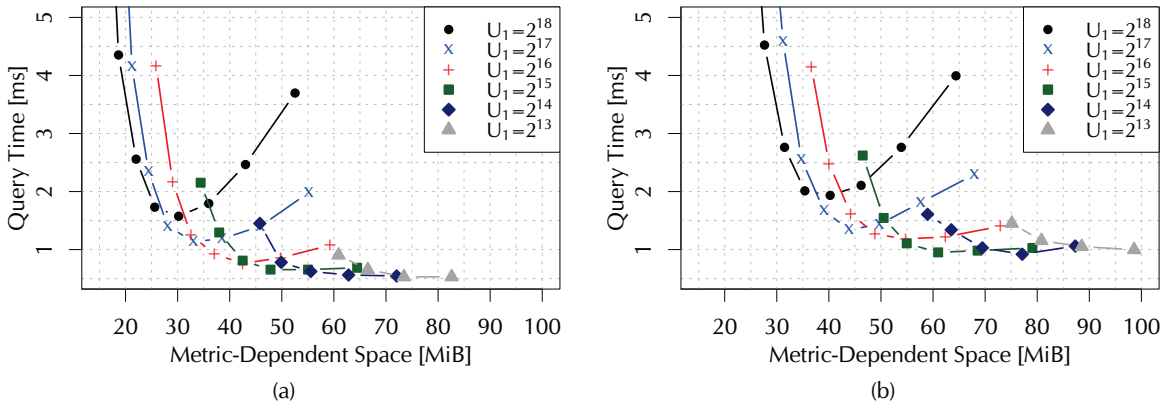


Figure 6.4. Performance of 2-level CALT with travel time and distance metric. For each line, U_1 is fixed and U_0 varies; the i-th data point from the right indicates $U_0 = 2^{7+i}$.

Dijkstra $\mathcal{O}(L)$ times to pick L vertices as landmarks, and stores distances between these landmarks and all vertices in the overlay. Queries use these distances and the triangle inequality to guide the search towards the goal. A complication of core-based approaches [GKW09, BDS+10] is the need to pick nearby overlay vertices as *proxies* for the source or target to get their distance bounds. Hence, queries use four CPU cores: Two pick the proxies, while two conduct the actual bidirectional search.

Evaluation. Figure 6.3 shows the query times and the metric-dependent space consumption for the basic algorithm, CALT (with 32 *avoid* landmarks [GKW09]), and PCD, with reduced cliques as overlay graphs. With some increase in space, both goal-direction techniques yield significantly faster queries (around one millisecond). PCD, however, needs much smaller cells and, thus, more space and customization time (about a minute for $U = 2^{14}$) than ALT (less than 3 s). Both methods are more effective for travel times than travel distances.

6.2.4. Multiple Levels

To accelerate queries, we can use multiple levels of overlay graphs, a common technique for partition-based approaches, including HiTi [JP02]. We need *nested partitions* of G, in which every boundary arc at level i is also a boundary arc at level $i-1$, for any $i > 1$. The level-0 partition is the original graph, with each vertex as a cell. See also Section 3.2 for more details on nested multilevel partitions. For the i-th level partition, we create a graph H_i as before: It includes all boundary arcs, plus an overlay linking the boundary vertices within a cell. Note that H_i can be computed bottom-up by reusing H_{i-1}. We use PUNCH to create multilevel partitions, in top-down fashion.

An s–t query runs bidirectional Dijkstra on a restricted graph G_{st}. An arc (u, v) from H_i will be in G_{st} if both u and v are in the same cell as s or t at level $i + 1$. Goal-direction can still be used on the top level. We call the resulting algorithm *Multilevel Dijkstra* (MLD).

Evaluation. Figure 6.4 shows the performance of the multilevel algorithm with two overlay levels (with reduced cliques) and ALT on the top level. We report query times and metric-dependent space for multiple values of U_0 and U_1, the maximum cell sizes on the bottom and top levels. A comparison with Figures 6.2 and 6.3 reveals that using two levels enables much faster queries for the same space. For travel times, a query takes 1 ms with about 40 MiB (with $U_0 = 2^{11}$ and $U_1 = 2^{16}$). Here, it takes 16 s to compute the bottom overlay, 5 s to compute the top overlay, and only 0.5 s to process landmarks. With 60 MiB space, queries take as little as 0.5 ms.

6.3. Streamlined Implementation

Although sparsification techniques save space and goal direction accelerates queries, the improvements are moderate and come at the expense of preprocessing time, implementation complexity, and metric-independence (the overlay topology is only metric-independent with full cliques). Furthermore, the time and space requirements of the simple clique implementation can be improved by representing each cell of the partition as a *matrix*, making the performance difference even smaller. The matrix contains the 32-bit distances among its entry and exit vertices (these are the vertices with at least one incoming or outgoing boundary arc, respectively; most boundary vertices are both). We also need arrays to associate rows (and columns) with the corresponding vertex IDs, but these are small and shared by all metrics.

We thus created a matrix-based *streamlined implementation* that is about twice as fast as the adjacency-based clique implementation. It does not use edge reduction, since it no longer saves space, slows down customization, and its effectiveness depends on the metric. (Skipping infinite matrix entries would make queries only slightly faster.) Similarly, we excluded CALT from the streamlined representation, since its queries are complicated and have high variance [BDS+10].

Phantom Levels. Customization times are typically dominated by building the overlay of the lowest level, since it works on the underlying graph directly (higher levels work on the much smaller cliques of the level below). As we have observed, smaller cells tend to lead to faster preprocessing. Therefore, as an optimization, the streamlined implementation includes a *phantom level* (with $U = 32$) to accelerate customization, but throws it away for queries, keeping space usage unaffected. For MLD-1 and MLD-2, we use a second phantom level with $U = 256$ as well.

Table 6.2. Performance of various algorithms for travel time and distance metrics.

	Travel Times				Distances			
	Customizing		Queries		Customizing		Queries	
Algorithm [Cell Sizes]	Time [s]	Space [MiB]	Vertex Scans	Time [ms]	Time [s]	Space [MiB]	Vertex Scans	Time [ms]
CALT [2^{11}:2^{16}]	21.3	37.1	5 292	0.92	17.2	48.9	5 739	1.26
MLD-1 [2^{14}]	4.9	10.1	45 420	5.81	4.8	10.1	47 417	6.12
MLD-2 [2^{12}:2^{18}]	5.0	18.8	12 683	1.82	5.0	18.8	13 071	1.83
MLD-3 [2^{10}:2^{15}:2^{20}]	5.2	32.7	6 099	0.91	5.1	32.7	6 344	0.98
MLD-4 [2^{8}:2^{12}:2^{16}:2^{20}]	4.7	59.5	3 828	0.72	4.7	59.5	4 033	0.79
CH economical	178.4	151.3	383	0.12	1 256.9	182.5	1 382	1.33
CH generous	355.6	122.8	376	0.10	1 987.4	165.8	1 354	1.29

Evaluation. Table 6.2 compares our streamlined multilevel implementation (called MLD, with up to 4 levels) with the original 2-level implementation of CALT. For each algorithm, we report the cell size bounds in each level. (Because CALT accelerates the top level, it uses different cell sizes than MLD-2.) We also consider two versions of Contraction Hierarchies (CH): The first (*economical*) minimizes preprocessing times, and the second (*generous*) the number of shortcuts [GSSV12]. For CH, we report the total space required to store the shortcuts (8 bytes per arc, excluding the original graph). For all algorithms, preprocessing uses four cores and queries use at least two.

We do not permute vertices after CH preprocessing (as is customary to improve query locality), since this prevents different metrics from sharing the same graph. Even so, with travel times, CH queries are one order of magnitude faster than our algorithm. For travel distances, MLD-3 and MLD-4 are faster than CH, but only slightly. For practical purposes, all variants have fast enough queries.

The main attraction of our approach is efficient metric customization. We require much less space: For example, MLD-2 needs about 20 MiB, which is less than 5 % of the original graph (more than 400 MiB) and an order of magnitude less than CH. Most notably, customization times are small. We need only five seconds to deal with a new metric, which is fast enough to enable personalized driving directions. This is two orders of magnitude faster than CH, even for a well-behaved metric. Phantom levels help here: Without them, MLD-1 would need about 20 s.

Note that CH customization can be faster if the processing order is fixed in advance [GSSV12]. The economical variant can rebuild the hierarchy (sequentially) in 54 s for travel times and 178 s for distances (still slower than our method). Unfortunately, using the order for one metric to rebuild another is only efficient if they are very similar [GKS10]. Also note that one can save space by storing only the upper part of the hierarchy [DSSW09a], at the expense of query times.

Table 6.2 shows that we can easily deal with real-time traffic: if all arc costs change (due to a traffic update), we can handle new queries after only five seconds. We can also support *local updates* quite efficiently. If a single arc cost changes, we must recompute at most one cell on each level, and MLD-4 takes less than a millisecond to do so. This is another reason for not using edge reduction or CALT: With either technique, changes in one cell may propagate beyond it.

6.4. Incorporating Turn Cost

So far, we have considered a simplified (but standard [DSSW09a]) representation of road networks, with each intersection corresponding to a single vertex. This is not very realistic, since it does not account for turn costs (or restrictions, a special case). Of course, any algorithm can handle turns simply by working on an expanded graph. A traditional [DSSW09a] representation is *arc-based*: Each vertex represents one *exit point* of an intersection, and each arc is a road segment followed by a turn. The expanded graph has one vertex for every road segment and one arc for every turn.

Compact Representation. The expanded graph is wasteful. We propose a *compact representation* in which each intersection becomes a single vertex with some associated information. If a vertex u has p incoming and q outgoing arcs, we associate a $p \times q$ *turn table* T_u to it, where $T_u[i, j]$ represents the turn from the i-th incoming arc into the j-th outgoing arc. Note that in our customizable setting, each entry should represent just a turn type (such as "left turn with stop sign"), since its cost may vary with different metrics. In addition to the turn tables, we store with each arc (u, v) its *tail order* (its position among u's outgoing arcs) and its *head order* (its position among v's incoming arcs). These orders may be arbitrary. Since degrees are small, 4 bits for each suffice.

In practice, many vertices share the same turn table. The total number of such *intersection types* is modest—in the thousands rather than millions. For example, many degree-four vertices in the United States have four-way stop signs. Each distinct turn table is, thus, stored only once, and each vertex keeps a pointer to the appropriate type, with little overhead.

Figure 6.5 illustrates different approaches to model turn costs. Besides the discussed models discussed, we also consider the *fully expanded* graph. Each entry and exit point becomes a vertex in this model, while arcs represent turns and road segments.

Augmenting Query Algorithms. With the compact turn representation, Dijkstra's algorithm becomes more complicated. In particular, it may now visit each vertex (intersection) multiple times, once for each entry point. It essentially simulates an execution on the arc-based expanded representation, which increases its running

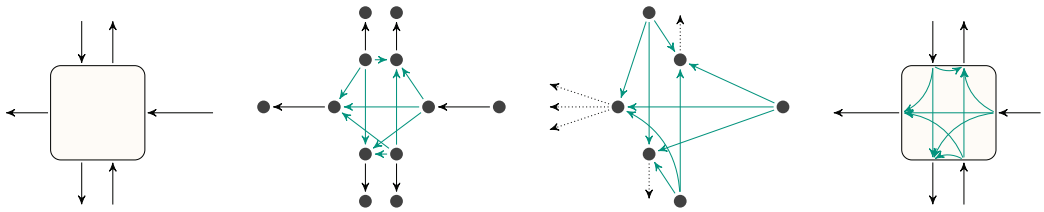


Figure 6.5. Turn representations (from left to right): none, fully expanded, arc-based, and compact.

time on Europe from 3 s to about 12 s. With a *stalling* technique, we can reduce the time to around 7 s. When scanning one entry point of an intersection, we can set bounds for its other entry points, which are not scanned unless their own distance labels are smaller than the bounds. These bounds depend on the turn table and can be computed during customization.

To support the compact representation, MLD needs two minor changes. First, it uses a turn-aware version of Dijkstra's algorithm on the lowest level (but not on higher ones). Second, matrices in each cell now represent paths between incoming and outgoing *boundary arcs* (and not boundary vertices, as before). The difference is subtle. With turns, the distance from a boundary vertex v to an exit point depends on whether we enter the cell from arc (u, v) or arc (w, v), so each arc needs its own entry in the matrix. Since most boundary vertices have only one incoming (and outgoing) boundary arc, the matrices are only slightly larger. (Note that, in essence, a cell can be seen as a giant turn table.)

Evaluation. We are not aware of publicly-available realistic turn data, so we augment our standard benchmark instance. For every vertex u, we add a turn between each incoming and each outgoing arc. A turn from (u, v) to (v, w) is either a *U-turn* (if $u = w$) or a *standard turn* (if $u \neq w$), and each of these two types has a cost. We have not tried to further distinguish between turn types, since any automated method would not reflect real-life turns. However, adding U-turn costs is enough to reproduce the key issue we found on realistic (proprietary) data.

Table 6.3 compares some algorithms on the European network augmented with turns. We consider two metrics, with U-turn costs set to 1 s or 100 s. The metrics are otherwise identical: Arc costs represent travel times and standard turns have zero cost. We tested four variants of MLD (with one to four levels) and two versions of CH (generous): *CH expanded* is the standard algorithm run on the arc-based expanded graph, while *CH compact* is modified to run on the compact representation. Column *vertex scans* counts the number of heap extractions.

Small U-turn costs do not change the shortest path structure of the graph much. Indeed, CH compact still works quite well: Preprocessing is only three times slower (than reported in Table 6.2), the number of shortcuts created is about the same, and queries take marginally longer. Using higher U-turn costs (as in a system that

Table 6.3. Performance of various algorithms on Europe with varying U-turn costs.

	U-turn: 1 s				U-turn: 100 s			
	Customizing		Queries		Customizing		Queries	
Algorithm [Cell Sizes]	Time [s]	Space [MiB]	Vertex Scans	Time [ms]	Time [s]	Space [MB]	Vertex Scans	Time [ms]
MLD-1 [2^{14}]	5.9	10.5	44 832	9.96	7.5	10.5	62 746	12.43
MLD-2 [2^{12}:2^{18}]	6.3	19.2	12 413	3.07	8.4	19.2	16 849	3.55
MLD-3 [2^{10}:2^{15}:2^{20}]	7.3	33.5	5 812	1.56	9.2	33.5	6 896	1.88
MLD-4 [2^{8}:2^{12}:2^{16}:2^{20}]	5.8	61.7	3 556	1.18	7.5	61.7	3 813	1.28
CH expanded	3 407.4	880.6	550	0.18	5 799.2	931.1	597	0.21
CH compact	846.0	132.5	905	0.19	23 774.8	304.0	5 585	2.11

avoids U-turns), however, makes preprocessing much less practical. Customization takes more than six hours, and space more than doubles. Intuitively, nontrivial U-turn costs are harder to handle because they increase the importance of certain vertices; for example, driving around the block may become a shortest path. Query times also increase, but are still practical. Note that recent independent work [GV11] shows that additional tuning can make compact CH more resilient: Changing U-turn costs from zero to 100 s increases customization time by a factor of only two. Unfortunately, forbidding U-turns altogether still slows it down by an extra factor of six.

With the expanded representation, CH preprocessing is much costlier when U-turns are cheap (since it runs on a larger graph), but is much less sensitive to an increase in the U-turn cost; queries are much faster as well. The difference in behavior is justified. While the compact representation forces CH to assign the same "importance" (order) to different entry points of an intersection, the expanded representation lets it separate them appropriately.

MLD is much less sensitive to turn costs. Compared to Table 6.2, we observe that preprocessing space is essentially the same (as expected). Preprocessing and query times increase slightly, mainly due to the lower level: High U-turn costs decrease the effectiveness of the stalling technique on the turn-enhanced graph.

In the most realistic setting, with nontrivial U-turn costs, customization takes less than 10 seconds on our commodity workstation. This is more than enough to handle frequent traffic updates, for example. If even more speed is required, one could simply use more cores: Speedups are almost perfect. On a server with two 6-core Xeon 5680 CPUs running at 3.33 GHz, MLD-4 takes only 2.4 seconds, which is faster than just running sequential Dijkstra on this input.

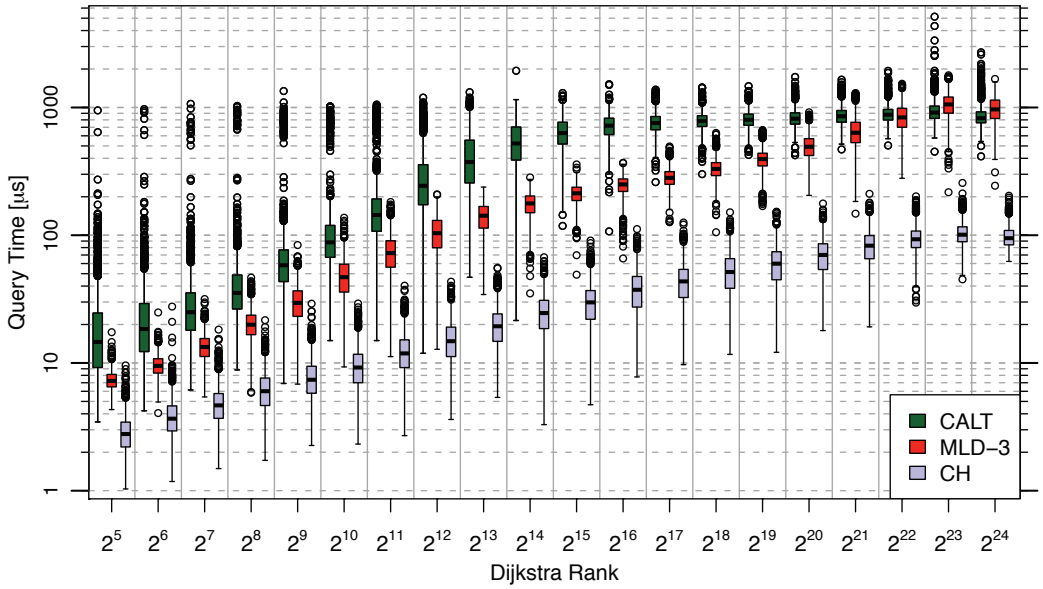


Figure 6.6. Performance of 2-level CALT, streamlined MLD-3, and CH for various query ranges (with travel times).

6.5. Further Experiments

This section contains further experiments for local queries and additional inputs.

Local Queries. In real-world applications most queries tend to be local. To evaluate our algorithm on queries of various ranges, Figures 6.6, 6.7, and 6.8 plot query times (without turns) subject to Dijkstra rank [SS05]. For a search from s, the Dijkstra rank of a vertex u is i if u is the i-th vertex scanned when Dijkstra's algorithm is run from s. We ran 1 000 queries per rank, with s chosen uniformly at random.

We observe that all algorithms are faster for local than for random queries. Only CALT has some slow outliers which is due to the two-phase nature of the query. Even local queries can be in different cells on the topmost level. In such a case the CALT query scans all vertices up to the topmost level, which is considerably slower than running MLD or CH. Adding a forth layer to MLD (Figure 6.8) accelerates queries on all scales.

Other Inputs. Table 6.4 reports the performance of CALT, MLD, and CH on two additional metrics (without turns). The first one is travel times *avoiding freeways* which penalizes the top three road categories, while the other one is *walking*. Here, we forbid highways and assign an average speed of 5 km/h to any other road segment, which we make undirected. We observe that the performance of all algorithms falls

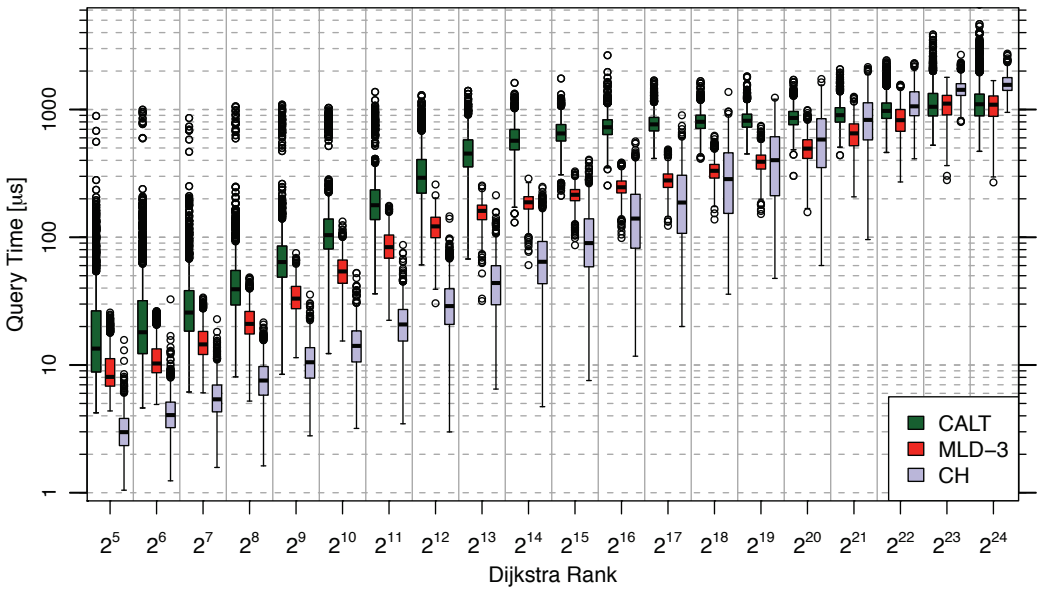


Figure 6.7. Performance of 2-level CALT, streamlined MLD-3, and CH for various query ranges (with distances).

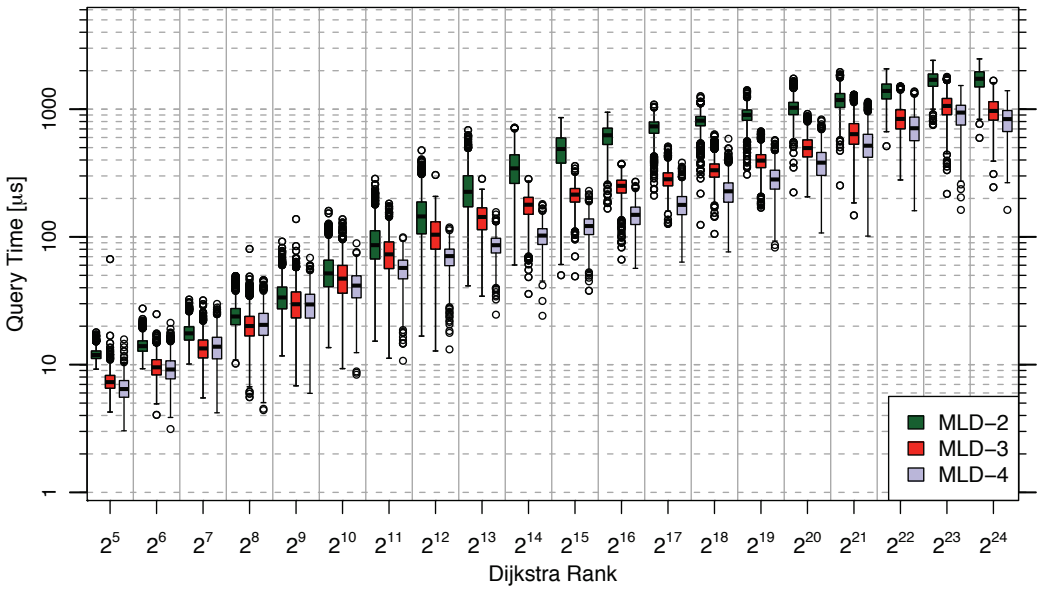


Figure 6.8. Comparison on MLD-2, MLD-3, and MLD-4 (travel times)

Table 6.4. Performance of various algorithms for walking metric and travel time metric that avoids highways.

	Avoid Freeways				Walking			
	Customizing		Queries		Customizing		Queries	
Algorithm [Cell Sizes]	Time [s]	Space [MiB]	Vertex Scans	Time [ms]	Time [s]	Space [MB]	Vertex Scans	Time [ms]
CALT [2^{11}:2^{16}]	19.7	38.8	5 519	1.12	18.1	44.6	5 724	1.21
MLD-1 [2^{14}]	6.4	9.8	45 958	5.73	4.7	9.8	45 982	5.77
MLD-2 [2^{12}:2^{18}]	6.0	12.6	12 936	1.71	5.0	12.6	12 847	1.73
MLD-3 [2^{10}:2^{15}:2^{20}]	5.6	32.3	6 231	1.20	5.0	32.3	6 130	0.95
MLD-4 [2^{8}:2^{12}:2^{16}:2^{20}]	5.1	59.0	4 041	0.77	4.9	59.0	3 939	0.75
CH economical	183.9	147.2	490	0.20	574.9	172.0	1 039	0.69
CH generous	367.4	119.6	489	0.18	973.4	152.5	1 037	0.71

Table 6.5. Performance of various algorithms for travel times and distances. The input is USA.

	Travel Times				Distances			
	Customizing		Queries		Customizing		Queries	
Algorithm [Cell Sizes]	Time [s]	Space [MiB]	Vertex Scans	Time [ms]	Time [s]	Space [MiB]	Vertex Scans	Time [ms]
MLD-1 [2^{14}]	8.1	14.0	52 007	7.29	7.9	14.0	54 226	7.34
MLD-2 [2^{12}:2^{18}]	8.2	26.2	13 317	1.77	8.0	26.2	13 680	1.88
MLD-3 [2^{10}:2^{15}:2^{20}]	8.3	47.1	6 031	1.25	8.2	47.1	6 236	1.33
MLD-4 [2^{8}:2^{12}:2^{16}:2^{20}]	8.2	94.4	5 708	1.17	7.7	94.4	5 976	1.27
CH economical	302.6	191.4	333	0.08	884.8	206.0	1 078	0.46
CH generous	580.8	163.1	294	0.07	1 646.3	186.0	1 072	0.46

between travel times and travel distances (cf. Figure 6.2).

Moreover, Table 6.5 shows the performance of MLD and CH on the (simplified, non-turn) road network of the United States of America (US). The graph has been made available for the 9th DIMACS implementation challenge on shortest paths [DGJ09]. It has 24 million vertices and 29 million road segments. We observe similar performance as for Europe. Using MLD-4, however, seems to pay off less: The metric-dependent space more than doubles compared to MLD-3, and queries are only slightly faster.

6.6. Path Unpacking

So far, we have reported the time to compute only the distance between two points. Following the parent pointers of the meeting vertex of forward and backward searches,

we may obtain a path containing shortcuts. To unpack a level-i shortcut, we run bidirectional Dijkstra on level $i-1$ (and recurse as necessary). Using all four cores, unpacking less than doubles query times, with no additional customization space. In contrast, standard CH unpacking stores the "middle" vertex of every shortcut, increasing the metric-dependent space by 50 %. For even faster unpacking, one can store a bit with each arc at level i indicating whether it appears in a shortcut at level $i+1$. This makes unpacking four times faster for MLD-2, but has little effect on MLD-3 and MLD-4.

6.7. Implementation Details

This section provides details on the implementation of the streamlined variant of MLD and on CH used in our experiments.

Streamlined MLD. Our non-turn implementation of MLD keeps an overlay graph with multiple levels. It contains n_o vertices, m_o arcs and $\sum_i s_i$ matrices, where s_i is the number of cells in level i. Each vertex, numbered from 0 to $n_o - 1$, can be an entry or exit point of a cell and hence stores its entry or exit position for each level, if it is a boundary vertex on this level. Moreover, it stores its ID in the original graph. Arcs are stored explicitly, while matrices are stored as arrays. We store topology and metric information separately.

Our query starts on the original graph but switches to the overlay graph at boundary arcs. We keep a hash table for each boundary vertex in the original graph. Then, when relaxing a boundary arc, we determine (via the hash table) the ID of the head vertex in the overlay graph. Switching from the overlay to the original graph is easy since we store the original ID for each overlay vertex explicitly. Note that our query algorithm is bidirectional. The backward search has to access the matrices in a cache-inefficient way. By also storing each matrix transposed, we can achieve additional speedups of 15 % but the metric dependent data doubles. Hence, we store each matrix only once.

There are only small differences between the non-turn and the turn-based implementations. In the latter, an overlay vertex is either an entry or an exit vertex. Hence, we additionally store the *type* (exit or entry vertex) of a vertex. The second difference is that we need to hash triples (u, p, t), where u is boundary vertex, p is the order of the incident cut arc (cf. Section 6.4), and t is the type. The query algorithm uses a turn-aware variant of Dijkstra's algorithm on the lower level and switches to a normal execution of Dijkstra's algorithm on the overlay graph.

Contraction Hierarchies. Our non-turn implementation follows the one presented in [GSSV12]. The differences are as follows. First, we use a different priority term during contraction. The priority of a vertex u is given by $2 \cdot \mathrm{ED}(u) + \mathrm{CN}(u) + \mathrm{H}(u) +$

$5 \cdot \mathrm{L}(u)$, where $\mathrm{ED}(u)$ is the difference between the number of arcs added and removed (if u were contracted), $\mathrm{CN}(u)$ is the number of contracted neighbors, $\mathrm{H}(u)$ is the total number of arcs represented by all shortcuts added, and $\mathrm{L}(u)$ is the level u would be assigned to. In this term, we bound $\mathrm{H}(u)$ such that every incident arc of u can contribute at most 3. This ensures that this term is only important during the beginning of the contraction process. Finally, our code is parallelized: After contracting a vertex, we update the priorities of all neighbors simultaneously. This gives a speedup of 2.5 over a sequential implementation.

Our generous and economical CH variants differ in how the witness searches are bounded. In our generous variant, witness searches are bounded by 5 hops up to an average degree (of the uncontracted graph) of 5, 10 hops up to degree 10, and no limit beyond that. The economical version uses 1 hop up to degree 3.3, 2 up to 10, 3 up to 10, and 5 beyond that.

We use the same set of parameters when running CH on turn-based inputs. When using the expanded graph, we can use our CH code without any modifications. On the compressed graph, however, the witness search becomes more complex. When contracting a vertex v, with incoming arcs (u_i, v) and outgoing arcs (v, w_j), we have to run searches from all entry vertices of all u_i to all exit vertices of all w_i. Besides that, the adaption is straightforward.

6.8. Conclusion

Recent advances in graph partitioning motivated us to reexamine the separator-based multilevel approach to the shortest path problem. With careful engineering, we drastically improved query speedups relative to Dijkstra's algorithm from less than 60 [HSW08] to more than 3000. With turn costs, the speedup increases even more, to 7000. This makes interactive queries possible. Furthermore, by explicitly separating metric customization from graph partitioning, we enable new metrics to be processed in a few seconds. The result is a flexible and practical solution to many real-life variants of the problem, such as real-time traffic and personalized cost functions. In fact, the algorithm presented in this chapter is, at the time of writing, the core of the routing engine in use by Bing Maps [Mic12].

Interesting open problems include adapting our approach to augmented scenarios, such as mobile or time-dependent implementations. In particular, a unidirectional version of MLD is also practical. Since partitions have a direct effect on performance, we would like to improve them further, perhaps by explicitly taking the size of the overlay graph into account.

Chapter 7

Computation of Jogging Routes

In this chapter, we study the problem of computing *jogging routes* in pedestrian networks. Given a source vertex s (the user's starting point), and a desired length L (in kilometers), the problem asks for a *cycle* of length (approximately) L that contains the vertex s. A "good" jogging route is, however, not only determined by its length; other criteria are just as important. An ideal route might follow paths through nice areas of the map (e. g., forests, parks, etc.), has rather circular shape, and not too many intersection at which the user is required to turn. A practical algorithm must, therefore, take all of these criteria into account.

Related Work. Much research focused on efficient methods for the related, but simpler, problem of computing point-to-point (shortest) paths. In fact, a plethora of algorithms exist, many of which are surveyed in [DSSW09a, Som12]. See Chapter 2 for an overview on the available literature. Speedup techniques usually employ sophisticated preprocessing to accelerate queries. In contrast, much less practical work exists for computing cycles. Graphs may contain exponentially many (in the number of vertices) cycles, even if they are planar [BKK+07]. If the length of the cycles is restricted by L, they can be enumerated in time $\mathcal{O}((n+m)(c+1))$, where c is the number of cycles of length at most L [LW06]. If one is interested in computing cycles with exactly k edges, the problem can be solved in $\mathcal{O}(2^k m)$ expected time [YZ95]. Unfortunately, none of these methods seems practical in our scenario. To the best of our knowledge, no efficient algorithm that computes sensible jogging routes exist.

Our Contribution. This chapter introduces the Jogging Problem. It turns out to be NP-hard, hence, we propose two heuristic approaches. The first, *Greedy Faces* is based on building the route by successively joining adjacent faces of the network. The second, *Partial Shortest Paths*, exploits the intuition of constructing equilateral polygons via shortest paths. The latter can be easily parallelized and has the inherent

property of providing sensible alternative routes. The result of our algorithms are routes of length within $(1 \pm \epsilon)L$, but also consider other important criteria that optimize the surrounding area, shape, and route complexity. An experimental study justifies our approaches: Using OpenStreetMap data, we are able to compute jogging routes of good quality in under 200 ms time; fast enough for interactive applications.

Overview and References. This chapter is organized as follows. Section 7.1 defines variants of the problem and shows NP-hardness. Section 7.2 introduces our two algorithmic approaches. Section 7.3 presents experiments, and Section 7.4 contains some concluding remarks.

The chapter is based on [Zün12] and [GPWZ13], the latter of which has been accepted at the 12th International Symposium on Experimental Algorithms (SEA'13). It is joint work with Andreas Gemsa, Dorothea Wagner, and Tobias Zündorf.

7.1. Problem Statement

Before we formally define the considered problems, we need to develop some notation (which somewhat differs from Section 3). We model pedestrian networks as *undirected graphs* $G = (V, E)$ with nonnegative integral *edge costs* $\ell\colon E \to \mathbb{Z}_{\geq 0}$. Usually, vertices correspond to intersections and edges to walkable segments. Also, we assume that our graphs admit straight-line embeddings, since vertices have associated latitude/longitude coordinates. For simplicity, our graphs are always connected. A *path* P is a sequence of vertices $P = [u_1, \ldots, u_k]$ for which $u_i u_{i+1} \in E$ must hold. Note that we sometimes just write u_1-u_k-path or P_{u_1,u_k} for short. If the first and last vertices coincide, we call P a *cycle*. The *cost* of a path, denoted by $\ell(P)$, is the sum of its edge-costs. A *shortest path* between two vertices u_1 and u_2 is a u_1-u_2-path with minimum cost. At some places we require intervals around a value $x \in \mathbb{Z}_{\geq 0}$ with error $\varepsilon \in \mathbb{R}_{\geq 0}$. We define them by $I(x, \varepsilon) = [\lfloor (1-\varepsilon)x \rfloor, \lceil (1+\varepsilon)x \rceil]$.

Simple Jogging Problem. The first problem we consider is the Simple Jogging Problem (SJP): We are given a graph G, source vertex $s \in V$, and a targeted cost $L \in \mathbb{Z}_{\geq 0}$ as input. The goal is to compute a cycle P through s with cost $\ell(P) = L$. In practical scenarios, cost usually represent geographical length. It turns out that SJP is NP-hard by reduction from Hamiltonian Cycle, which we prove in the following theorem.

Theorem 4. *The* Simple Jogging Problem *is NP-hard.*

Proof. We show NP-hardness of SJP by reduction from the problem Hamiltonian Cycle, which is known to be NP-hard [Kar72]: Given an undirected graph $G = (V = \{u_1, \ldots, u_n\}, E)$ with n vertices and m edges, it asks for a cycle P that contains each vertex from V exactly once. From G, we now construct a new graph $G' = (V', E')$, on which a solution of SJP corresponds to a Hamiltonian cycle in G (and vice versa).

Note that solutions of SJP may, in general, contain vertices multiple times. We, therefore, make the reduction work by carefully defining edge costs $\ell\colon E \to \mathbb{Z}_{\geq 0}$ of G' with respect to an appropriate base b. We set b to $2n+1$ and construct G' as follows. We duplicate each vertex $u_i \in V$ to $\text{in}_i, \text{out}_i \in V'$ and create an edge in_iout_i with cost $\ell_{\text{vert}}(i) = b^n + b^i$. Moreover, each edge $u_iu_j \in E$ is represented in a' by adding the edge in_iout_j with constant cost $\ell_{\text{edge}} = b^n$. We set the source vertex s to out_0 and the requested length $L = n \cdot \ell_{\text{edge}} + \sum_i \ell_{\text{vert}}(i)$.

Now, if G contains a Hamiltonian cycle $P = [u_{i_0}, \ldots, u_{i,n}]$, the jogging route $P' = [\text{out}_{i_0}, \text{in}_{i_1}, \text{out}_{i_1}, \ldots, \text{in}_{i_n}, \text{out}_{i_n}]$ in G' is also feasible. Its length $\ell(P')$ is $\sum_j(\ell_{\text{edge}} + \ell_{\text{vert}}(j))$, which by definition equals L.

On the other hand, assume that P' is a jogging route of length L in G'. From the way we chose edge costs, it follows that P' must have exactly $2n+1$ vertices (i. e., $2n$ edges): The shortest route with more than $2n+1$ vertices must be longer than L, and the longest route with less than $2n+1$ vertices must be shorter than L. The definition of L encodes the cost of each edge in_iout_i exactly once. From this, we also know that P' must contain each edge in_iout_i exactly once. The vertices of the Hamiltonian cycle P are, then, given in order defined by their corresponding vertices in P'. ■

Note that from this, NP-hardness follows for the respective optimization problem, i. e., finding a cycle that *minimizes* $|\ell(P) - L|$.

Dynamic Program for SJP. If we allow running time in the order of L, one can solve SJP by a dynamic program, similarly as it is known for the Subset Sum Problem [GJ79]. The algorithm maintains a boolean matrix $Q\colon V \times \mathbb{Z}_{\geq 0} \to \{0,1\}$ of size $|V| \times L$, which indicates whether a path to vertex u with cost ℓ exists. Initially, Q is set to all-zero, except for the entry $Q(s,0)$, which is set to 1. It then considers subsequent cost values ℓ in increasing order (beginning at 0). In each step, the algorithm checks for all edges $uv \in E$ if an existing path can be extended to v with cost ℓ. It does so by looking if $Q(u, \ell - \ell(uv))$ is set to 1, updating $Q(v,\ell)$ accordingly. The algorithm stops as soon as ℓ exceeds the input cost L. Then, the requested jogging route exists iff $Q(s,L) = 1$ holds. The running time of the algorithm is $\mathcal{O}(L|E|)$ and, thus, we conclude that the SJP is weakly NP-hard [GJ79].

Relaxed Jogging Problem. In practice, solely optimizing length (or cost) may result in undesirable routes. Jogging is a recreational activity, therefore, one usually also considers the surrounding area (parks and forests), the shape (preferably edge-disjoint), and the complexity of the route (small number of turns). We argue that the primary goal remains geographical length. However, we allow some (user-specified) slack on the length to take the aforementioned criteria into account. This motivates the Relaxed Jogging Problem (RJP): Given a graph G, a source vertex $s \in V$, input length $L \in \mathbb{Z}_{\geq 0}$, and a parameter $\epsilon \in [0,1]$, the goal is to compute a cycle P through s

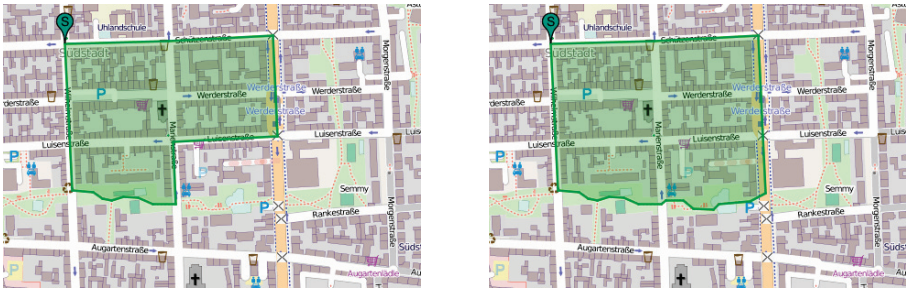


Figure 7.1. Illustrating the intuition of the Greedy Faces approach: A tentative jogging route (left) is extended by attaching one of its adjacent blocks (right).

with cost $\ell(P) \in I(L, \varepsilon)$ while optimizing a set of *soft criteria*. We identify three important criteria in the following.

To account for the surrounding area, we introduce *badness* as a mapping on the edges $\text{bad}\colon E \to [0,1]$. Smaller values indicate "nicer" areas (e. g., parks). Badness values on the edges are provided by the input data. To extend badness to paths, we combine it with the path's length. (Note that we assume costs to represent geographical length for the remainder of this work.) That is, for a path $P = [u_1, \ldots, u_k]$ its badness is defined by $\text{bad}(P) = \sum \text{bad}(u_i u_{i+1}) \ell(u_i u_{i+1}) / \ell(P)$. By these means, badness values are scaled by their edge lengths, but are still in the interval $[0,1]$. This enables comparing paths (wrt. badness) of different lengths.

To optimize edge-disjointness of paths, we consider *sharing*. It counts edges that appear at least twice on P, scaled by their length. Formally, it first accumulates into a set D all indices i, j for which either $u_i u_{i+1} = u_j u_{j+1}$ or $u_i u_{i+1} = u_j u_{j-1}$ hold. (Note that edges are undirected.) The sharing of path P is then $\text{sh}(P) = \sum_{i \in D} \ell(u_i u_{i+1}) / \ell(P)$. Sharing values are also in $[0,1]$.

To evaluate route complexity, we consider *turns*. For two edges a and b, we measure their angle $\measuredangle(a,b)$ and regard them as a turn, if and only if $\measuredangle(a,b) \notin I(180°, \alpha)$ holds. We usually set α to 15 %.

7.2. Algorithms

We now introduce our two approaches for the Relaxed Jogging Problem: *Greedy Faces* and *Partial Shortest Paths*. We present each approach in turn, starting with a basic version, then, proposing optimizations along the way.

7.2.1. Greedy Faces

Assume that we are already given a tentative jogging route (i. e., a cycle in G that contains s). A natural way to extend it, is to attach one of its adjacent "blocks" that

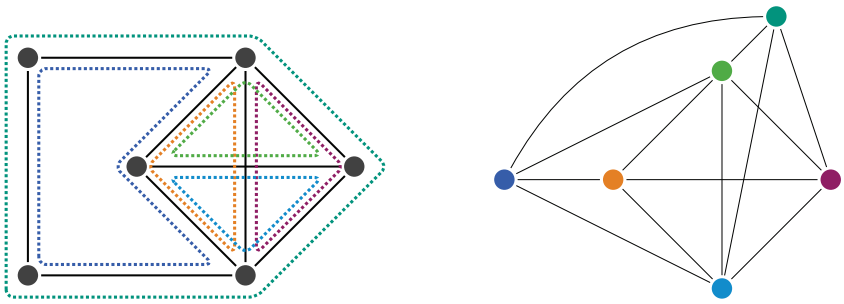


Figure 7.2. Illustrating duality: Faces of G (left) are represented by vertices in G^* (right).

lie on the "outer" side of the route. Then, repeat this step, until a route of desired size and shape has been grown. See Figure 7.1 for the intuition.

In a planar graph, blocks correspond to faces. But our inputs may contain intersecting edges (such as bridges and tunnels), albeit only few in practice. We, therefore, propose preprocessing G to identify blocks (we still call them faces). These are used by our greedy faces algorithm. Finally, we present smoothening techniques to reduce route complexity in a quick postprocessing step.

Identifying Faces. For our algorithm to work, we must precompute a set F of *faces* in G. We identify each face $f \in F$ with its *enclosing* path P_f. Our preprocessing involves several steps. First, we delete the *1-shell* of G by iteratively removing vertices (and their incident edges) from G that have degree one. The resulting graph is 2-connected and no longer contains dead-end streets (which we want to avoid, anyway). Next, we consider all remaining edges $uv \in E$. For each, we perform a *right-first search*, thereby, constructing an enclosing path P_f for a new face f. More precisely, we run a depth-first search, beginning at uv. Whenever it reaches a vertex x (via an edge a), it identifies the unique edge b that follows a in the (counterclockwise) circular edge ordering at x. (Note that this ordering is always defined for embedded graphs.) It adds b to P_f. If $b = uv$, the algorithm stops and adds f to F, discarding duplicates. However, since G is not necessarily planar, the edge b might intersect with one of its preceding edges on P_f. In this case, it removes b from P_f and considers the next edge (after b) in the circular order at x for expansion. While constructing F, the algorithm remembers for each edge a list of its incident faces. It uses them to build a *dual graph* $G^* = (V^*, E^*)$: Vertices correspond to faces (of G), and two faces are connected in G^*, iff they share at least one edge in G. This definition of G^* extends the well-known graph duality for planar graphs, however, as G may not be planar, so may not be G^*. An example of two corresponding graphs G and G^* is illustrated in Figure 7.2. The running time of the preprocessing is dominated by the face-detection step. For every edge it runs a right-first search, each in time $\mathcal{O}(|E|)$. Whenever it expands an edge, it must perform intersection tests with up to $\mathcal{O}(|V|)$

preceding edges. This results in a total running time of $\mathcal{O}(|V||E|^2)$. Note that we expect much better running times in practice: On realistic inputs we may assume faces to have constant size.

Greedy Faces Algorithm. Our greedy faces algorithm, short GF, now uses G^* as input. Its basic idea is to run a (modified) breadth first search (BFS) on G^*. It starts by selecting an arbitrary face $f \in V^*$ that contains the source vertex s, i. e., where $s \in P_f$ holds. It then grows a BFS-tree T (rooted at f), until a stopping condition is met. When it stops, the jogging route P is retrieved by looking at the set of *cut edges* that separate T from $V^* - T$: Their corresponding edges in G constitute a cycle. (Note that this is a well-known property on planar graphs, but carries over to our definition of G^*.) However, to make P a feasible jogging route, we must ensure two properties: The cycle must be (a) simple and (b) still contain s. We ensure both while growing T. Regarding (a), we know that the corresponding cycle P in G is simple iff the subgraph induced by $V^* - T$ is connected. We check this condition when expanding an edge $fg \in E^*$ during the BFS, discarding fg if adding g to T would disconnect $V^* - T$. Regarding (b), The vertex s is still part of the jogging route as long as at least one incident face of s remains in $V^* - T$. We also perform this check while expanding edges, discarding them whenever necessary. The result of every iteration of the BFS is a potential jogging route P. The algorithm stops as soon as the cost of P exceeds $(1+\epsilon)L$. It then returns, among all discovered routes whose length is in $I(L, \epsilon)$, the one with minimum total badness.

However, up to now, GF does not *optimize* badness. To guide the search towards "nice" areas of the graph, we propose a force-directed approach. Therefore, consider a face f and the *geometric center* $C(f)$ of its enclosing path. Inspired by Newton's law of gravity, we define a force vector $\vec{\phi}(f, p)$ acting upon a point p of the map by

$$\vec{\phi}(f, p) = (\text{bad}(f) - 0.5)\ell(f)/|\vec{d}|^2 \cdot \vec{d}/|\vec{d}|, \tag{7.1}$$

where $\vec{d} = p - C(f)$. Note that, depending on $\text{bad}(f)$, the force is repelling or attracting. Also, the vector $\vec{\phi}(f, p)$ is directed, and its intensity decreases with the distance squared. Now, the force that acts upon a face g is the sum of the forces over all (other) faces in the graph (toward g). More precisely, $\vec{\phi}(g) = \sum_{f \in V^*} \vec{\phi}(f, C(g))$. In practice, we quickly precompute these values restricted to reachable faces (i. e., faces within a radius of $L/2$ from s). The BFS in our algorithm now extends the edge $fg \in E^*$ next, for which g has the highest force in *direction of extension*. More precisely, it extends fg, iff g maximizes the term $\vec{\phi}(g)\cos(\measuredangle(\vec{\phi}(g), C(f) - C(P)))$. Note that $C(P)$ is the geometric center of the current (tentative) jogging route P in the algorithm, and $\measuredangle(\cdot, \cdot)$ measures the angle of two vectors. Figure 7.3 illustrates force-direction. In principle, further criteria can be added to the BFS (e. g., via linear combinations): The *roundness* considers the ratio of the route's perimeter to its

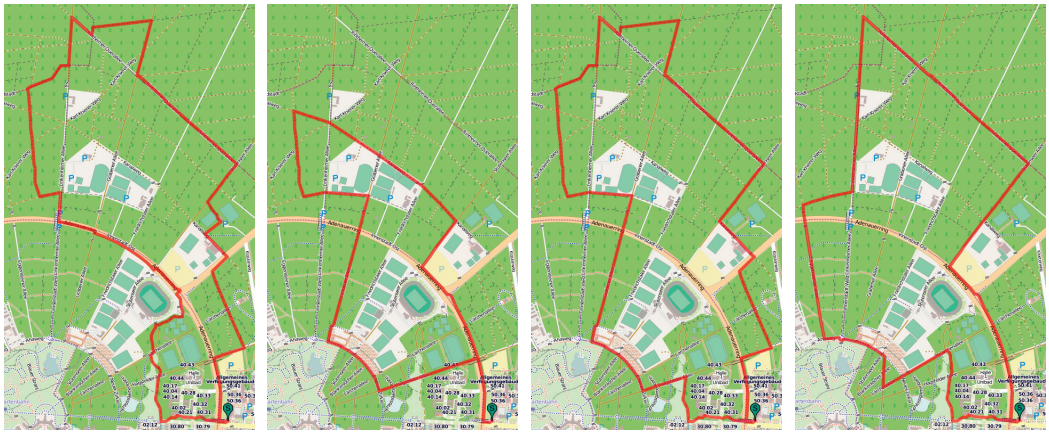
(a) ℓ: 3303 m, bad: 75 %. (b) ℓ: 3036 m, bad: 43 %.

Figure 7.3. Two example jogging routes obtained by the basic GF algorithm without (left) and with (right) force-direction (we set $L = 3000\,\mathrm{m}$ and $\epsilon = 20\,\%$). Note that force-direction successfully finds the nearby forest.

area (lower values are better); *convexity* takes the distance between a candidate face and the current route into account (higher values are better). However, preliminary experiments showed that (on realistic inputs) the effect of these criteria is limited. The running time of GF is bounded by the BFS on G^*. In the worst case, it scans $\mathcal{O}(|V^*|)$ faces. The next face it expands to can be determined in time $\mathcal{O}(|V^*|)$, yielding a total running time of $\mathcal{O}(|V^*|^2)$. Finally, recall that our preprocessing removes the 1-shell of G. For the case that the source vertex s is part of the 1-shell, we quickly find the (unique) path P' to the first vertex s' that is not in the 1-shell. We then run our algorithm, but initialized with s' and $L' = L - 2\ell(P)$, simply attaching P' to the route afterward. Also note that routes obtained by GF are optimal with respect to sharing: The only (unavoidable) place it may occur is on P' (in case s is in the 1-shell).

Route Smoothening. By default, GF provides no guarantee on route *complexity* (i. e., on the number of turns). We, therefore, propose reducing it by *smoothing* the route in a postprocessing step. To do so, we first select a small subsequence $P' \subset P$ of the route's vertices. (Note that s must be part of P'.) Then, for each two subsequent vertices $uv \in P'$, we compute a shortest u-v-path (e. g., by Dijkstra's algorithm [Dij59]). Finally, concatenating these paths produces the smoothened route. To also take badness into account, we use a custom metric $\omega\colon E \to \mathbb{Z}_{\geq 0}$, defined by $\omega(a) = \mathrm{bad}(a)\ell(a)$, when computing shortest paths.

It remains to discuss how we choose the subsequence P' from P. We propose three rules. The first, called *equidistant rule* (es), simply selects the k (an input parameter) vertices from P, which are distributed equally regarding their subsequent distances. More precisely, vertex $u \in P$ is selected as the i-th vertex on P' if it minimizes $\ell(P)i/k - \ell(P_{s,u})$ (here, $P_{s,u}$ denotes the subpath of P up to vertex u). Unfortunately, this rule may select vertices at arbitrary (with respect to the route's shape) positions. Therefore, our second rule, called *convex rule* (cs), obtains P' by



(a) ℓ: 8533 m, bad: 28 %. (b) ℓ: 8181 m, bad: 29 %. (c) ℓ: 8181 m, bad: 29 %. (d) ℓ: 8181 m, bad: 29 %.

Figure 7.4. Example query from the computer science department in Karlsruhe with $L = 8000\,\text{m}$ and $\epsilon = 10\,\%$. We use the GF algorithm without smoothening (a), opposed to smoothening by the equidistant rule (b), convex rule (c), and important vertex rule (d).

computing the *convex hull* of P, e. g., by running Graham's Scan algorithm [Gra72] on P. In case the source vertex s is not part of the convex hull, we must still add it to P': We set its position next to the first vertex of P that is contained in P''s convex hull. Finally, the third rule, called *important vertex rule* (ivs), tries to identify k (again, an input parameter) "important" vertices of P: At first, it slices P into k subpaths of equal length. From each, it then selects the vertex u whose incident edges have lowest total badness (i. e., $\prod_{uv \in E} \text{bad}(uv)$ is minimized). This rule follows the intuition that vertices that share many edges of low badness are more likely in "nicer" areas. See Figure 7.4, which compares the effect of these rules. Note that while smoothening helps to reduce route complexity, its drawback is that the route's length may change arbitrarily. We address this issue by our next approach.

7.2.2. Partial Shortest Paths

As discussed, GF provides no guarantee on the deviation from the requested route length, if they are smoothened. We, therefore, propose a second approach: It directly computes a set of *via vertices*, connected by shortest paths, but such that the length of the resulting routes is guaranteed to be in $I(L, \epsilon)$. In the following, we refer to jogging routes that use k via vertices by *k-via-routes*.

2-via-routes. For our basic version, we exploit the intuition of constructing equilateral triangles (see Figure 7.5, left, which illustrates the principle), thus, obtaining 2-via-routes. We know that s must be part of the route. Therefore, we choose s

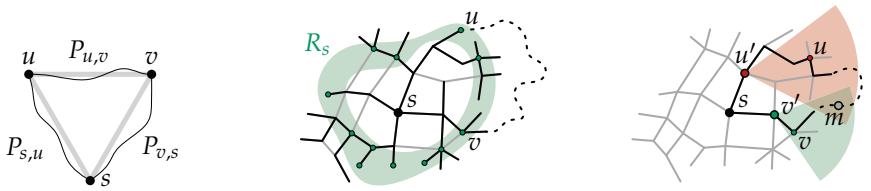
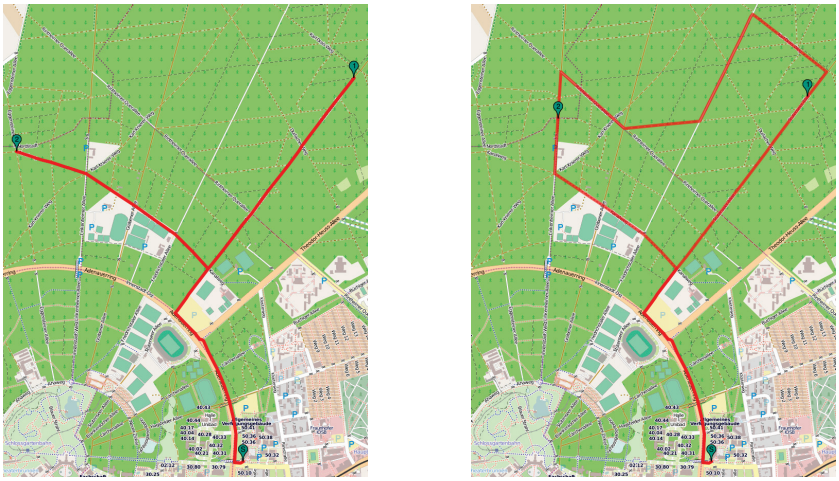


Figure 7.5. Left: Intuition of constructing 2-via-routes. Middle: Shortest path tree rooted at s and ring R_s with candidate vertices u, v forming a feasible route (dotted). Right: Selecting middle vertices m that lie "behind" u, v in the shortest path trees of u', v'.

as one of the triangle's vertices. It now remains to compute two vertices u, v (and related paths), such that $\ell(P_{s,u}), \ell(P_{u,v}), \ell(P_{v,s}) \in I(L/3, \epsilon)$. From this, we obtain the required total length of $I(L, \varepsilon)$. To select u and v, we, at first, define a metric on the edges $\omega\colon E \to \mathbb{Z}_{\geq 0}$ that takes the edge's badness into account. As in Section 7.2.1, we set $\omega(a) = \mathrm{bad}(a)\ell(a)$. We now run a shortest path computation on G from s using this metric with Dijkstra's algorithm [Dij59]. To limit the search, we do not relax edges out of vertices x for whom $\ell(P_{s,x})$ exceeds $(1+\epsilon)L/3$. (Note that $\ell(P_{x,s})$ can be stored with x during the algorithm with negligible overhead.) The resulting shortest path tree T_s (rooted at s) accounts for "nice" paths by optimizing ω and provably contains all feasible candidate vertices u (and v). We refer to this subset of candidate vertices as *ring* around s with distance $I(L/3, \epsilon)$, in short R_s. We must now find two vertices of the ring that have a connecting path with length $I(L/3, \epsilon)$. To do so, we pick a vertex u from the ring R_s and compute *its* ring R_u (also with respect to length $I(L/3, \epsilon)$) by running Dijkstra's algorithm from u, similarly to before. Now, the intersection of R_s with R_u exactly contains the matching vertices v, that is, concatenating $P_{s,u}, P_{u,v}, P_{v,s}$ yields an admissible jogging route (i. e., of length $I(L, \epsilon)$). See Figure 7.5 (middle) for an illustration. The algorithm repeats this step for all vertices in R_s, and it selects, among all admissible routes it discovers, the one minimizing badness. We call this algorithm PSP2 (partial shortest paths with two vias). We remark that distances other than $L/3$ are possible when computing rings. This varies the route's shape and corresponds to constructing "triangles" with nonuniform side lengths. The running time of PSP2 is dominated by up to $\mathcal{O}(|V|)$ shortest path computations, thus, it is bounded by $\mathcal{O}(|V|^2 \log |V| + |V||E|)$. Note that we expect much better performance in practice, as the shortest path computations are local.

We now propose two optimizations for PSP2. First, the algorithm can be sped up by a *stopping criterion*. For it to work, it must pick vertices u from R_s in order of increasing value $\omega(P_{s,u})$. Note that this order is automatically provided by Dijkstra's algorithm. It then only needs to consider paths $P_{v,s}$ as third leg of the route, for whom $\omega(P_{v,s}) \geq \omega(P_{s,u})$ holds (all others have been evaluated earlier). By this, the total badness of any route P the algorithm may still find is lower-bound by $\mathrm{bad}_{\mathrm{lb}} = 2\omega(P_{s,u})/(1+\epsilon)L$. If we keep track of the route P_{opt} minimizing badness, the algorithm may stop as soon as $\mathrm{bad}_{\mathrm{lb}}$ exceeds $\mathrm{bad}(P_{\mathrm{opt}})$—it will provably not find any route with lower badness.



(a) ℓ: 8533 m, bad: 28 %. (b) ℓ: 8181 m, bad: 29 %.

Figure 7.6. Example query for PSP2 from the computer science department in Karlsruhe with $L = 8000\,\text{m}$ and $\epsilon = 10\,\%$. Reduced sharing is disabled on the left and enabled on the right. The markers represent the vertices s, u, and v. Note that the left output is an extreme case regarding sharing—not the average case.

Up to now, PSP2 has no guarantee on the sharing of P. In fact, it can be up to 100 % in extreme cases, thus, we propose the following optimization. When the algorithm computes R_u for a vertex $u \in R_s$, we forbid it to relax any edges from $P_{s,u}$. This ensures that $P_{s,u}$ and $P_{u,v}$ are edge-disjoint. To also make $P_{u,v}$ and $P_{v,s}$ edge-disjoint, we disregard routes whose last edges of $P_{u,v}$ and $P_{v,s}$ coincide. Note that we still allow sharing wrt. to the first and last legs of the route (around s). See Figure 7.6 for examples.

3-via-routes. Jogging routes obtained by PSP2 follow shortest paths for each of its three legs $P_{s,u}$, $P_{u,v}$, and $P_{v,s}$. However, no such guarantee exists around u and v, which might be undesirable. We now propose an optimized variant of our algorithm, PSP3. It aims to smoothen the route around u and v. Moreover, it uses three via-vertices, which, in general, produces more circular shaped routes.

The algorithm follows the intuition of constructing regular *quadrilaterals*. Taking the source vertex s as one of the quadrilateral's vertices, it must therefore compute vertices u, m, and v, connected by paths $P_{s,u}$, $P_{u,m}$, $P_{m,v}$, and $P_{v,s}$, each with length $I(L/4, \epsilon)$. We refer to m as *middle vertex*. The algorithm starts, again, by first computing a ring R_s of vertices from s, but now with distance $I(L/4, \epsilon)$. (It does so by using Dijkstra's algorithm with metric ω.) To smoothen the route around u and v, we do not use u and v directly as sources for the subsequent shortest path computations (like we did with PSP2). Instead, we consider the (tighter) ring R'_s of vertices

(a) ℓ: 8538 m, bad: 28 %. (b) ℓ: 8080 m, bad: 29 %.

Figure 7.7. Example query for PSP3 with $L = 8000\,\text{m}$ and $\epsilon = 10\,\%$. Sharing is avoided around m on the right (as opposed to the left).

around s with distance $I(\alpha L/4, \epsilon)$. Here, the parameter α takes values from $[0.5, 1]$ and controls smoothness around u and v. We obtain the ring R'_s by traversing the shortest path tree from each vertex $u \in R_s$ upward, until the distance condition is met. Moreover, the vertex u remembers which vertex u' it created in R'_s (this is required later). Next, the algorithm picks vertices u' from R'_s (in any order) and computes, for each, a ring $R_{u'}$ around u'. To account for α, we set the distance of $R_{u'}$ to $I((2-\alpha)L/4, \epsilon)$. It follows that vertices in $R_{u'}$ have distance $I(L/2, \epsilon)$ from s, containing potential middle vertices. Having computed all rings, we then consider for each pair of vertices u', v' in R'_s the intersection M of their rings, i. e., $M = R_{u'} \cap R_{v'}$. The algorithm now selects only such middle vertices $m \in M$ that result in smooth paths around u and v. More precisely, a vertex $m \in M$ is selected, iff the *smoothing condition* holds, i. e., the path $P_{u',m}$ contains u and the path $P_{v',m}$ contains v. Intuitively, we are only interested in the part of M that lies "behind" u (resp. v) on the shortest path tree of $R_{u'}$ ($R_{v'}$). See Figure 7.5 (right) for an illustration. Each vertex m that fulfills the smoothing condition represents an admissible jogging route by concatenating $P_{s,u}$, $P_{u,m}$, $P_{m,v}$, and $P_{v,s}$. The algorithm returns, among those, the one with minimum badness. With PSP3, the only vertex around which sharing may occur is m (besides s). We avoid it by discarding middle vertices m, for which the last edges of $P_{u',m}$ and $P_{v',m}$ coincide. This can be efficiently checked during the algorithm. See Figure 7.7 for an example.

Optimizations. We now propose two optimizations to speed up PSP3. The first avoids the costly computation of set-intersections: Instead of storing (and intersecting) rings $R_{u'}$, the algorithm maintains a vertex-set M_m at each vertex m of the graph. Whenever Dijkstra's algorithm scans a potential middle vertex m, it adds u to M_m (iff the smoothing condition holds). Moreover, it suffices to keep the (at most) two vertices u, v with lowest associated badness values in each set M_m. As a result, managing middle vertices is a constant time operation. The second optimization avoids some calls to Dijkstra's algorithm: If the ring R'_s contains vertices u' and v' for which u' is an ancestor of v' in the shortest path tree, a single Dijkstra run from u' suffices to handle both u' and v'. Including these optimizations, PSP3 essentially runs $\mathcal{O}(|V|)$ times Dijkstra's algorithm. Its total running time is thus $\mathcal{O}(|V|^2 \log |V| + |V||E|)$, as well as PSP2's.

Bidirectional Search. To allow more flexibility for selecting the middle vertex, we propose the algorithm PSP3-Bi which is an extension of PSP3 using bidirectional search [Dan62]. As PSP3, it starts by computing R_s and, from that, R'_s. However, it now runs (in turn) for each *pair* of vertices u', v' a bidirectional search. Whenever it scans a vertex m that has already been scanned by the opposite direction, it checks (a) whether u (resp. v) are ancestors of m in the forward (resp. backward) shortest path tree, and (b) if the total length of the combined route is in $I(L, \epsilon)$. If both hold true, it stops and considers the just-found jogging route as output (it keeps track of the one that minimizes badness). Note that by design, sharing around m cannot occur. Since PSP3-Bi must run a bidirectional search for each pair of vertices in R'_s, its running time is bounded by $\mathcal{O}(|V|^3 \log |V| + |V|^2|E|)$.

Parallelization. All PSP-based algorithms can be easily parallelized in a shared memory setup: They, first, sequentially compute the ring R_s (resp. R'_s). Subsequent Dijkstra runs may then be distributed among the available processors. Each processor computes its locally optimal route, and the globally optimal route is selected in a sequential postprocessing step. To avoid race conditions, we use locking as synchronization primitive, whenever necessary.

Alternative Routes. All PSP-based algorithms provide *alternative routes* without significant computational overhead. Instead of just outputting the route with minimum badness, we may output the k best routes. However, these routes tend to be too similar. We, therefore, only consider routes as alternatives that are pairwise different in their via-vertices u and v from R_s (still selecting the k best regarding badness). By these means, we obtain jogging routes that cover different regions of the graph around the source vertex s. See Figure 7.8 for an example.

(a) ℓ: 7600 m, bad: 20 %. (b) ℓ: 7654 m, bad: 24 %. (c) ℓ: 7697 m, bad: 27 %.

Figure 7.8. Example query using PSP3-Bi with $L = 8000$ m and $\epsilon = 10$ %. The figure shows the best three (from a total of twelve found) alternative routes.

7.3. Experiments

We implemented all algorithms from Section 7.2 in C++ compiled with GCC 4.7.1 and flag `-O3`. Experiments were run on one core of a dual 8-core Intel Xeon E5-2670 clocked at 2.6 GHz with 64 GiB of DDR3-1600 RAM. We focus on the pedestrian network of the greater Karlsruhe region in Germany. We extracted data from a snapshot of the freely available OpenStreetMap (OSM) [Ope04] on 5 August 2012. We only keep walkable street segments and use OSM's `highway` and `landuse` (of the surrounding polygon, if available) tags to define sensible badness values. They are listed in Table 7.1. From these values, the badness of vertices $u \in V$ is defined by $\text{bad}(u) = \text{bad}(\text{landuse}(u))$. Furthermore, the badness for edges $a = \{u, v\} \in E$ is defined according to

$$\text{bad}(a) = \begin{cases} \text{bad}(\text{highway}(a)) & \text{if highway}(a) = \text{track}, \\ \frac{1}{2}\,\text{bad}(\text{highway}(a)) + \frac{1}{2}\max\{\text{bad}(u), \text{bad}(v)\} & \text{otherwise.} \end{cases}$$

The resulting graph has 104 759 vertices and 118 671 edges.

Evaluating Algorithms. Our first experiment evaluates quality and performance of our algorithms. For each, we ran (the same) 1 000 queries with source vertex s chosen at random. We request routes of 10 km length and ϵ set to 10 %. Results are summarized in Table 7.2. We report the average length (in km) of the computed routes, the standard deviation (Std.-Dev.) of their length, their average badness

Table 7.1. Badness values used in our experiments for different values of the `landuse` and `highway` tags in the OpenStreetMap data.

Tag `landuse`	Badness
allotments	0.5
brownfield	1
cemetery	1
commercial	1
construction	1
farm	0.2
farmland	0.2
farmyard	0.3
forest	0.1
garages	1
grass	0.15
greenfield	0.1
greenhouse_horticulture	0.6
industrial	1
landfill	1
leisure	0.15
meadow	0.1
military	1
orchard	0.5
plant_nursery	0.6
quarry	1
railway	0.5
recreation_ground	0.2
reservoir	0.3
residential	0.8
retail	1
unclassified	1
village_green	0.2
vineyard	0.4

Tag `highway`	Badness
bridleway	0.6
crossing	0.6
cycleway	0.2
footway	0.5
ford	1
living_street	0.7
path	0.5
pedestrian	0.8
residential	0.9
road	0.8
secondary	1
secondary_link	1
service	0.9
steps	0.5
tertiary	1
tertiary_link	1
track	0.15
track, grade1	0.1
track, grade2	0.15
track, grade3	0.25
track, grade4	0.35
track, grade5	0.45
unclassified	0.9

Table 7.2. Solution quality and performance on our Karlsruhe input for both the Greedy Faces (GF) and Partial Shortest Paths (PSP) algorithms. For smoothening, we apply the equidistant rule (es), convex hull rule (cs), and important vertex rule (ivs) to GF.

Algorithm	Length [km]	Std.-Dev.	Bad. [%]	Sh. [%]	No. Trn.	Succ. Rate	Time-1 [ms]	Time-4 [ms]	Time-8 [ms]
GREEDY FACES:									
GF	9.89	0.58	48.7	0.2	51	93 %	285	—	—
GF-es	9.61	2.07	43.8	6.5	28	93 %	289	—	—
GF-cs	9.73	2.23	43.0	6.9	29	93 %	296	—	—
GF-ivs	9.48	1.98	41.7	6.0	30	93 %	293	—	—
PARTIAL SHORTEST PATHS:									
PSP2	9.99	0.58	27.3	52.5	16	98 %	179	84	63
PSP3	10.14	0.41	31.0	23.6	20	98 %	155	78	72
PSP3-Bi	10.06	0.53	33.4	13.9	21	98 %	446	177	140

values (Bad.), their average amount of sharing (Sh.), the number of turns on them (No. Trn.), and the average running time of the algorithm on one, and, where applicable, also on four and eight processors (Time-x). Sometimes our algorithms may not find any feasible solution. Therefore, we also report their success rates (Succ. Rate).

Algorithms in Table 7.2 are grouped into blocks. The first evaluates the greedy faces approach from Section 7.2.1. We observe that GF succeeds in approximating the required route length of 10 km with very little error. However, for 7 % of our queries no solution was found. One reason is that GF is unable to recover from local optima. However, sharing is almost nonexistent with an average value of 0.2 %. This is expected, since by design sharing for GF only occurs around s, iff it lies in a dead-end street. On the downside, route complexity is quite high with 51 turns on average. This justifies our smoothening rules by shortest paths. We set the number of selected vertices to 6 for GF-es and to 9 for GF-ivs. Interestingly, figures are quite similar for all rules: They reduce route complexity by a factor of almost two, which comes with little increase in sharing (up to 6.9 %). Recall that smoothening may arbitrarily change route lengths. Our experiments indicate that the average route length deviates little (it is still 9.5–9.7 km, depending on the specific rule). However, the figure is much less stable: The mean error (Std.-Dev.) increases to around 2 km. Regarding running times, GF runs in 285 ms on average, with a mild increase up to 296 ms (≈ 4 %), if we enable smoothening.

The second block evaluates the PSP approach from Section 7.2.2 (we set α to 0.6, where applicable). Again, we succeed approximating the required route length of 10 km with little error (≈ 0.5 km on average for all algorithms). Because PSP considers more route combinations than GF, it is more likely to find a feasible solution. This is reflected by the excellent success rate of 98 % (for all PSP algorithms).

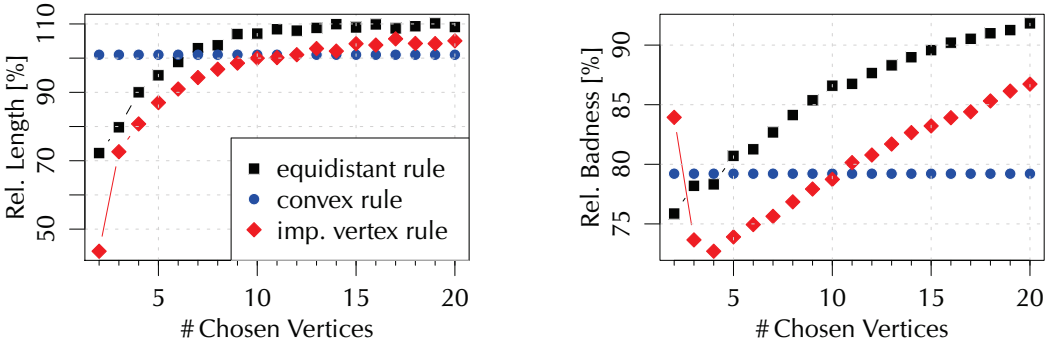


Figure 7.9. Evaluating the effect of the smoothening rules on GF. We report the relative amount by which the route's length (left) and badness (right) change while varying the number of vertices the algorithm selects to compute shortest paths (cf. Section 7.2.1). The key of the left figure also applies to the right.

Regarding badness, PSP finds "nicer" routes (lower average badness) than any of the GF algorithms. However, their sharing (still only possible around s) is much higher. On average, sharing is 52 % for PSP2's, though, we are able to reduce it to 14 % with PSP3-Bi. This is well acceptable in practice. An important advantage of PSP over GF is route complexity: With 16–21 turns on average, this figure is lower than *any* of the GF algorithms, even with applied smoothening. Enabling the stopping criterion decreases running times from 3 579 ms (not reported in the table) to 179 ms, a factor of 20. The fastest algorithm is PSP3 with 155 ms on average. PSP3-Bi is slower by a factor of 2.9. (Recall that it must run a bidirectional search for every *pair* of vertices from R_s; cf. Section 7.2.2.) Regarding parallelism, we observe speedups of factor 2.1 (PSP2) and 1.9 (PSP3) on four processors over a sequential execution. As expected, with a speedup of 2.5, PSP3-Bi benefits most from parallelization. Increasing the number of processors to eight, improves little. Still, PSP3-Bi benefits most, with a total speedup of 3.1.

Detailed Experiments. We now present two detailed experiments. The first concerns our smoothening rules, the second evaluates variations of the input parameter ϵ. Each datapoint is based on (the same) 1 000 queries with s selected at random and L set to 10 km. Figure 7.9 shows results of our first experiment. We set ϵ to 10 % and vary (on the abscissa) the number of vertices between which the smoothening process computes shortest paths. The left plot reports, for each smoothening rule, how much it affects the length of the routes. We report the average amount (in percent) it changes. The right plot shows the same figure, but for badness. We observe that our routes tend to get shorter after smoothening. This is expected, since we rebuild routes using shortest paths. Selecting too few vertices shortens routes severely (to below 50 %). Their length eventually stabilizes above 90 % for six vertices and more. Badness generally improves when using smoothening, but continuously

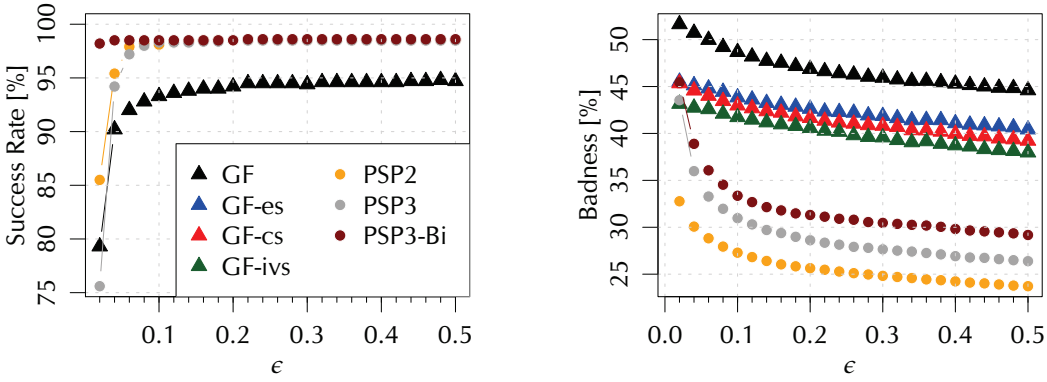


Figure 7.10. Evaluating success rate and badness on all algorithms for varying ϵ. The key of the left figure also applies to the right. Note that, regarding the greedy faces approach, smoothening does not affect the success rate, hence, we only report it for GF.

increases with more vertices. Interestingly, the convex rule (which is independent of the number of vertices) seems good regarding both length and badness, which makes it the preferred rule in practice.

Our final experiment evaluates all algorithms for varying input parameter ϵ. Results are summarized in Figure 7.10, which evaluates, for each ϵ, the average success rate (left plot) and the resulting route's badness (right plot). Note that applying smoothening to GF does not affect the success rate, therefore, we do not enumerate smoothening rules in the left figure. We observe that too much restriction on the allowed length (small ϵ-values), may result in a low success rate (down to 75 %) and high badness values (more than 50 % for GF). Setting $\epsilon > 0.07$ already significantly improves the success rate. Unsurprisingly, badness values gradually improve with increasing ϵ, as this gives the algorithms more room for optimization. Here, a good tradeoff seems setting ϵ to 0.1. Interestingly, PSP3-Bi's success rate is almost unaffected by ϵ, even for tiny values below 0.07.

Additional Input. Here we provide an experimental evaluation of our algorithms on another input. We use the greater region of Paderborn (Germany), which we (also) extracted from OpenStreetMap (OSM) data on 5 August 2012. Again, our graph only keeps walkable street segments with badness values defined using OSM's `highway` and `landuse` tags (see Table 7.1 for details). The resulting graph has 28 381 vertices and 33 340 edges.

Table 7.3 summarizes quality and performance results. It reports the same figures as Table 7.2 from Section 7.3. For each algorithm, they are based on running 1 000 queries with randomly selected source vertex s, 10 km length, and ϵ set to 10 %. We observe that, regarding quality, results are very similar: All algorithms approximate the requested length of 10 km with small error on average (with the exception of ap-

Table 7.3. Evaluating solution quality and performance on our Paderborn input for both the Greedy Faces (GF) and Partial Shortest Paths (PSP) algorithms. We report the same figures as in Table 7.2.

Algorithm	Length [km]	Std.-Dev.	Bad. [%]	Sh. [%]	No. Trn.	Succ. Rate	Time-1 [ms]	Time-4 [ms]	Time-8 [ms]
GREEDY FACES:									
GF	9.83	0.58	70.9	0.6	54	93 %	110	—	—
GF-es	8.98	2.11	68.1	9.4	28	93 %	112	—	—
GF-cs	8.97	2.01	67.7	6.3	27	93 %	113	—	—
GF-ivs	9.10	1.86	65.8	9.0	31	93 %	112	—	—
PARTIAL SHORTEST PATHS:									
PSP2	9.95	0.56	53.6	50.6	20	98 %	476	222	121
PSP3	10.04	0.41	58.0	20.0	25	98 %	58	29	26
PSP3-Bi	9.99	0.53	60.0	12.9	23	97 %	154	62	48

plied smoothening for GF). Also, sharing and the number of turns show similar (with respect to Table 7.2) figures for all algorithms. Interestingly, badness values are consistently higher than on Karlsruhe. Indeed, the city of Karlsruhe has many parks and is surrounded by more green areas, which allows for more jogging routes with lower badness. Success rates are, again, almost identical to Table 7.2: GF succeeds in in 93 % of the queries to find a solution, while PSP improves success rates to 97–98 %. Regarding running time, we observe that GF computes solutions in 110 ms time. Like on Karlsruhe, applying smoothening has almost no effect on running time. PSP2, on the other hand, is slower (by a factor of 2.6) than on Karlsruhe, but admits better parallel speedups (factor of 2.2 on four, and factor of 3.9 on eight cores). However, PSP3 is faster by a factor of 8.2 over PSP2. (Recall that PSP3 computes shortest paths from a tighter ring R'_s with less vertices; cf. Section 7.2.2). Consistently with Table 7.2, PSP3-Bi's running time increases over PSP3 by a factor of 2.6.

Case Study. We conclude our experimental evaluation with a case study of all algorithms. We evaluate them for three queries on the Karlsruhe input with varying lengths of 8 km, 12 km, and 5 km. The computed routes are presented in Figures 7.11, 7.12, and 7.13/7.14, respectively.

Regarding our first example, Figure 7.11a shows the output of PSP3-Bi, which seems to compute a generally attractive route. Thereby, its length is only exceeded by 38 m and the number of turns is 11, making the route relatively easy to remember. Recall that PSP2 generally computes routes with higher sharing (cf. Table 7.2). This is also depicted by Figure 7.11c. Note that in this example this may be beneficial: The route is able to use the quickest path through the (less pleasant) urban area to

enter and exit the nearby forest. The GF approach produces less attractive results. We observe the routes have significantly more turns. Even though this can be remedied by the smoothening rules to some extent, in doing so, the algorithm loses the property that it approximates the requested route length.

Looking at the second example in Figure 7.12, we observe a similar behavior of our algorithms as in the previous example (Figure 7.11). The PSP algorithms tend to produce superior results when compared to the GF approach. In particular, the unsmoothened route in Figure 7.12f) has way too many turns to be practical. Also note the slight difference of PSP3 and PSP3-Bi in Figures 7.12a and 7.12b, respectively. Recall that PSP3 does not guarantee that the path at the middle vertex m is locally optimal, which is reflected by the small detour around m in Figure 7.12a. In contrast, this detour is avoided in the output of PSP3-Bi.

The last example, depicted in Figure 7.13 and 7.14, is somewhat more challenging for our algorithms. Instead of a large forest close to the source vertex, it contains only a small green strip along the river Alb (which is, in fact, a very popular jogging area in Karlsruhe). We observe that, still, all algorithms succeed in finding this green area. However, PSP3 struggles to remain in the park for the entire route. Interestingly, in this example the performance of GF with smoothening enabled (Figures 7.13d and 7.14a) improves compared to the previous examples. Nonetheless, the length of the smoothened routes deviates significantly from the request input of $L = 5\,\text{km}$.

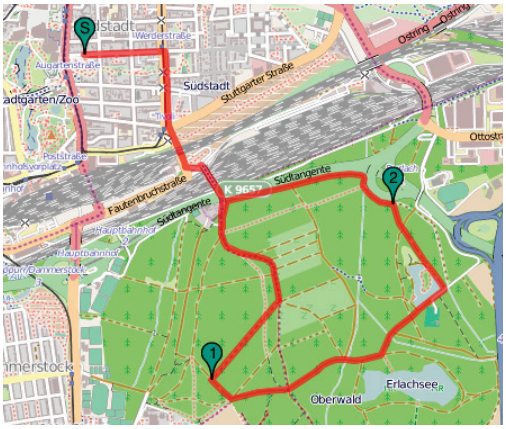
7.4. Conclusion

In this chapter we introduced the NP-hard Jogging Problem. To compute useful jogging routes, we presented two novel algorithmic approaches that solve a relaxed variant of the problem. Besides length, both explicitly optimize two important criteria: Badness (i. e., surrounding area) and sharing (i. e., shape of the route). The methods are based on different intuitions. The first incrementally extends routes by carefully joining adjacent faces of the graph, possibly smoothened by a quick postprocessing step. The second computes sets of alternative routes that resemble equilateral polygons via shortest path computations. Experiments on real-world data reveal that our algorithms are indeed practical: They compute jogging routes of excellent quality in under 200 ms time, which is fast enough for interactive applications.

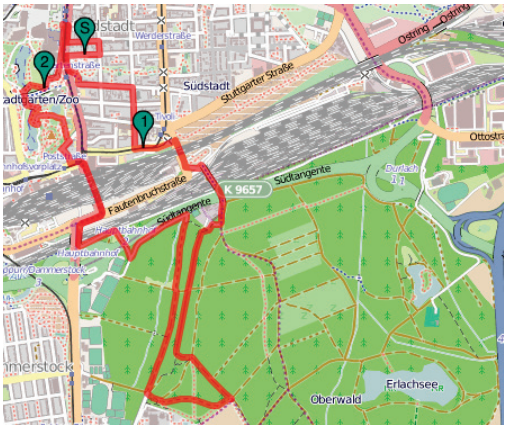
Future work includes comparing our algorithms to exact solutions and better methods for selecting via vertices—either as smoothening rules or for computing routes directly. Also, providing via vertices (or "areas") as input is an interesting scenario. Finally, we like to accelerate our algorithms further. Especially, PSP may benefit from speedup techniques [DSSW09a,Som12]. This, however, requires adapting them to compute rings instead of point-to-point paths.

(a) PSP3-Bi. ℓ: 8038 m, bad: 35 %, turns: 11.

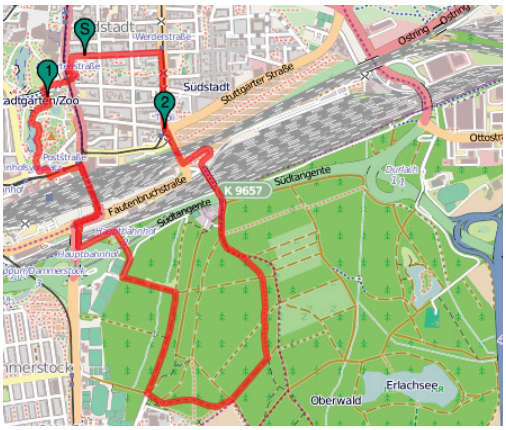
(b) PSP3. ℓ: 7996 m, bad: 35 %, turns: 14.



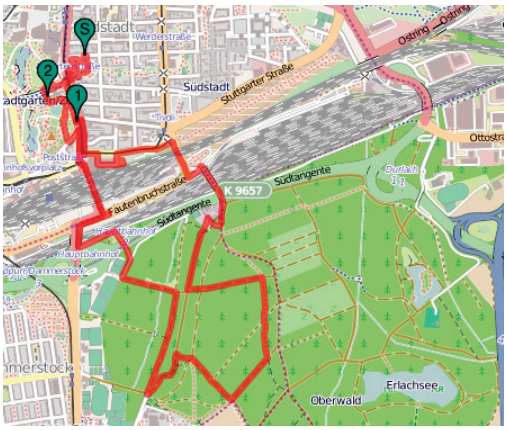
(c) PSP2. ℓ: 7923 m, bad: 34 %, turns: 15.



(d) GF-ivs. ℓ: 7767 m, bad: 43 %, turns: 43.

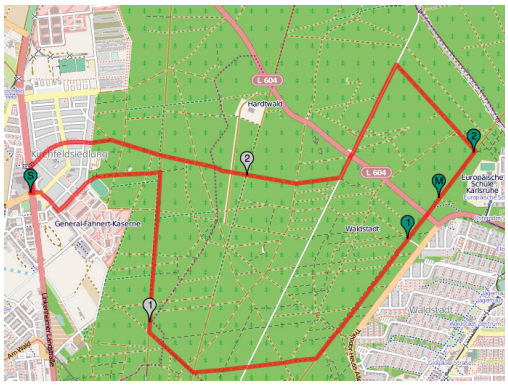


(e) GF-cs. ℓ: 6637 m, bad: 44 %, turns: 28.



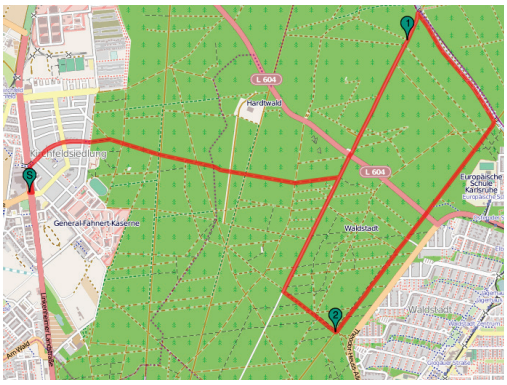
(f) GF. ℓ: 7883 m, bad: 44 %, turns: 39.

Figure 7.11. First example. Output of our algorithms for $L = 8$ km and $\epsilon = 10$ %.

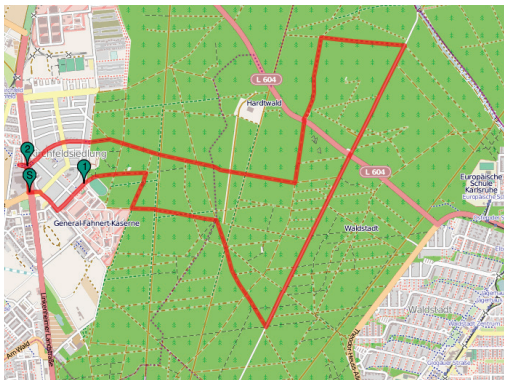


(a) PSP3-Bi. ℓ: 11 687 m, bad: 23 %, turns: 12.

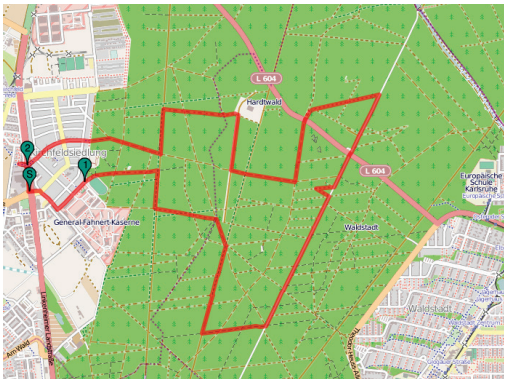
(b) PSP3. ℓ: 11 822 m, bad: 23 %, turns: 15.



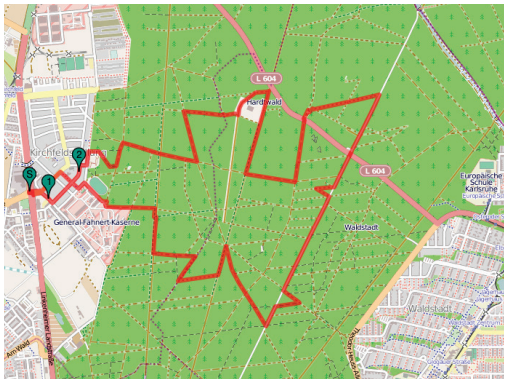
(c) PSP2. ℓ: 12 753 m, bad: 22 %, turns: 11.



(d) GF-ivs. ℓ: 10 811 m, bad: 27 %, turns: 14.



(e) GF-es. ℓ: 11 129 m, bad: 30 %, turns: 23.



(f) GF. ℓ: 12 254 m, bad: 29 %, turns: 40.

Figure 7.12. Second example. Output of our algorithms for $L = 12\,\text{km}$ and $\epsilon = 10\,\%$.

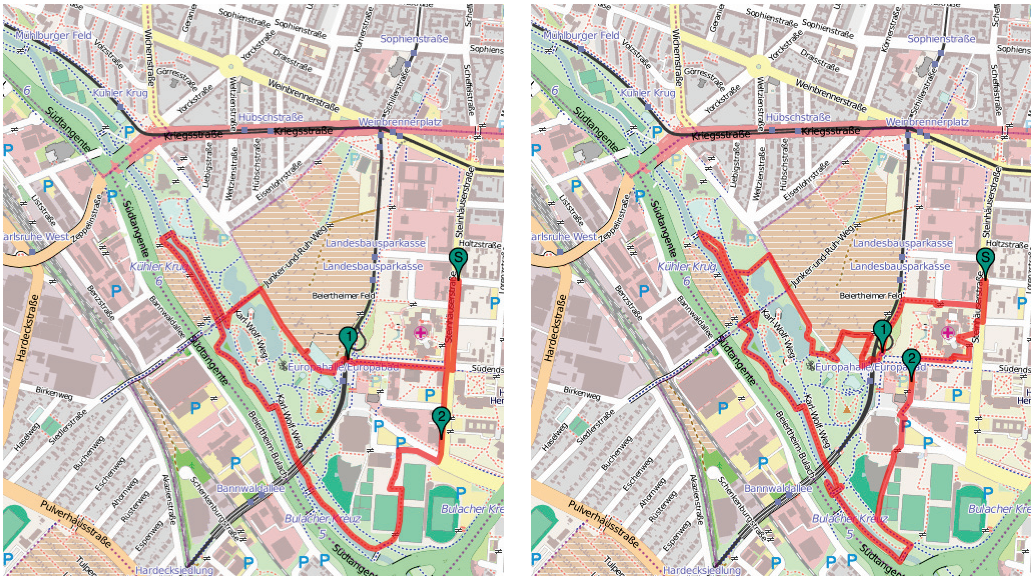
(a) PSP3-Bi. ℓ: 4755 m, bad: 30 %, turns: 16.

(b) PSP3. ℓ: 4769 m, bad: 48 %, turns: 22.

(c) PSP2. ℓ: 4838 m, bad: 30 %, turns: 18.

(d) GF-ivs. ℓ: 3987 m, bad: 37 %, turns: 21.

Figure 7.13. Third example. Output of our algorithms for $L = 5\,\text{km}$ and $\epsilon = 10\,\%$.



(a) GF-es. ℓ: 3982 m, bad: 40 %, turns: 19.

(b) GF. ℓ: 4620 m, bad: 41 %, turns: 44.

Figure 7.14. Third example, continued. Output of our algorithms for $L = 5\,\text{km}$ and $\epsilon = 10\,\%$.

Chapter 8

Conclusion

In this work, we developed new algorithms for four scenarios: Journey planning in public transit networks, journey planning in multimodal networks, customizable route planning in road networks, and computing jogging routes in pedestrian networks. We developed, evaluated, and refined our algorithms by following the paradigm of *Algorithm Engineering*: It is captured by a cycle of design, analysis, implementation, and experimentation. (Also recall Figure 1.1.) Thereby, we based a strong focus on real world applicability. We ran our experiments on real data and developed our algorithms with the underlying hardware architecture in mind. Both aspects turned out to be very important. We observed that the problems studied in this work are generally memory bound: They require to scan huge amounts of data, but with relatively little amount of performed computation per data item. As such, an essential condition to obtain fast practical algorithms is to carefully design the memory layout of the data structures that are maintained by the algorithm. Keeping the memory hierarchy of the underlying hardware architecture in mind makes a significant difference when it comes to the performance of the algorithms. This strengthens our belief that bringing theory and (realistic) experimentation together is vital for algorithmic research, if we aim to design efficient and provably correct algorithms that are intended for practical applications.

The remainder of this chapter briefly summarizes the main results for the problems we have studied and gives a general direction for interesting future work.

Public Transit Journey Planning. Most of the available journey planning methods build a graph from the input (i. e., the timetable) on which they run a shortest path algorithm that computes journeys. Thereby, the size of this graph has a direct impact on the performance of the algorithm. To this extent, we analyzed the realistic time-dependent model and refined it into a new Coloring Model for computing earliest arrival and range queries. By merging certain vertices that are not "in conflict", we were able to reduce the number of vertices by up to a factor of 12 (depending on the

input). Note that this accelerates *any* query algorithm. Moreover, we presented a heuristic for generating artificial footpaths. It utilizes the underlying road network to identify and interconnect stops that are close to the same intersection. Note that footpaths (often missing from the input data) are crucial to enable realistic queries.

We then developed an efficient range query algorithm that is based on our new Coloring Model, called SPCS. It systematically exploits the observation that in public transit networks the number of relevant connections to travel to the target can be limited in advance and that certain connections are dominated by others along the way. Unlike previous algorithms, which are notoriously hard to parallelize, we showed that our algorithm admits a natural and efficient parallelization. Moreover, by utilizing the very same algorithm to precompute journeys between important stops, we further accelerated point-to-point queries by up to a factor of four.

Inarguably important when planning public transit journeys is to consider the number of transfers taken. We, therefore, addressed the problem of computing Pareto sets of journeys that optimize arrival time and the number of transfers as criteria. To this extent, we presented RAPTOR, an algorithm that is neither graph-based, nor uses a priority queue. Instead, it efficiently organizes the timetable data into a few simple arrays. On these, it operates in rounds (one per transfer) and scans each route at most once per round. Our experiments on the full metropolitan network of London revealed that RAPTOR is more than an order of magnitude faster than previous approaches. Moreover, we showed how RAPTOR can be easily parallelized, further accelerating queries. We also extended RAPTOR to handle strict domination, additional criteria, and to compute multicriteria range queries. Since RAPTOR does not rely on preprocessing, it can be directly used in dynamic scenarios, easily handling delays and trip cancellations.

Multimodal Journey Planning. The second part of this thesis considered the problem of computing multimodal journeys in heterogenous networks consisting of walking, bicycles, car travel, public transit, and flights. We pointed out that a crucial task of any journey planner is to combine the available modes of transports in a *sensible* way.

An elegant approach to this problem is computing label-constrained shortest paths. To this extent, we presented UCCH: It augments Contraction Hierarchies to label-constrained shortest path computation in a sound manner, and as a result, it is the first multimodal preprocessing-based method that handles arbitrary mode sequence constraints as an input to the *query*—a problem considered challenging before. Besides that, since UCCH does not require a dedicated algorithm for local queries, it is simpler than previous algorithms. At the same time, when compared to previous approaches with similar query performance, UCCH handles multimodal networks with significantly denser public transit with less preprocessing effort.

Going from here, we extended RAPTOR to optimize multicriteria journey planning in multimodal networks. Instead of considering label-constraints, we argued that

users optimize, in addition to arrival time, mode-dependent convenience criteria, e. g., walking duration (walking), number of transfers (public transit), and monetary cost (taxi). We presented MCR, a novel multicriteria algorithm that computes exact Pareto sets of multimodal journeys. Since these Pareto sets are large, we used fuzzy set theory to extract a concise and diverse subset of the most significant journeys in a quick postprocessing step. To further accelerate our approach, we presented heuristics (still multicriteria) that closely match the best journeys of the exact Pareto set. We verified our approach on the full multimodal network of London, enabling efficient realistic multimodal journey planning in large metropolitan areas.

Customizable Route Planning in Road Networks. In this part of the thesis we considered the customizable route planning problem in road networks: Its goal was a technique that supports fast metric updates and has a query algorithm that is robust with respect to *any* metric. After carefully analyzing previous work, we revisited a method that is based on overlay graphs. Thereby, our approach splits preprocessing into a metric-independent stage and a customization stage. The first computes the topology of the overlays (using graph partitioning) and may take minutes or hours. By storing the overlays efficiently in memory as matrices, the customization stage can incorporate an arbitrary new metric within seconds. Our experiments on the continental network of Europe revealed that our approach is indeed highly practical: Metric customization takes seconds with very little space overhead (per metric) and our query algorithm is fast enough for interactive scenarios. This enables new applications, e. g., supporting personalized cost functions and real-time traffic data.

Computing Jogging Routes. The final part of this thesis dealt with the problem of computing jogging routes in pedestrian networks. To the best of our knowledge, we presented the first practical algorithmic solutions to this problem. The problem asks for a cyclic route that optimizes its shape, its vicinity, and its complexity, while approximating a given (as input) length. We showed that a formalization of the problem is weakly NP-hard. Nevertheless, we developed two general approaches to compute sensible jogging routes heuristically: The first, Greedy Faces, incrementally extends the route by attaching adjacent faces of the graph, possibly smoothening the route in a postprocessing step. The second approach, Partial Shortest Paths, is based on the intuition of constructing equilateral polygons. It can even be parallelized and inherently computes diverse sets of alternatives routes. We justified our algorithms by experiments on real-world data, showing that we are able to compute sensible jogging routes fast enough for interactive applications.

8.1. Future Work

In this section, we discuss interesting future work.

Public Transit Journey Planning. An interesting question deriving from our work on RAPTOR is whether we can develop a speedup technique based on RAPTOR. Recall that the performance of RAPTOR is bounded by the number of routes it scans per round. While practical in our experiments on the network of London, this number may be too high for networks beyond metropolitan scale. Large-scale networks, such as that of a whole country, usually consist of many (presumably loosely interconnected) local networks and an overlaying long-distance (train) network. Intuitively, optimal (even multicriteria) journeys from, e. g., Karlsruhe to Berlin should only use the respective local networks of Karlsruhe and Berlin. A first approach could be to look at (multilevel) overlay techniques [SWW00, SWZ02, JP02, DGPW11]. They should be helpful to distinguish routes that are important to travel *through* a cell versus routes that are important for travel *within* a cell. A long-distance query could then skip over local routes of intermediate cells.

Realistic public transit networks are notoriously prone to delays, cancellations, and other types of uncertainty. It would therefore be an interesting question how uncertainty can be incorporated into RAPTOR (beyond reliability). A first approach that minimizes *expected arrival time* (under probabilistic delays) is the Connection Scan Algorithm [DPSW13], however, it must look at the entire input for each query and, hence, does not scale. Maybe we can combine CSA and RAPTOR to obtain an algorithm that exploits (like RAPTOR) the notion of routes in a principled way in order to obtain fast and more scalable queries.

Another very interesting problem is to compute a more comprehensive output than individual journeys for public transit queries. Often, for the same values of, e. g., arrival time and number of transfers, many different journeys exist that vary in the stop where the user can transfer or even in the routes the user may take. On the other hand, in dense urban networks, exact departure and arrival times are of little value for some routes (think of a subway line that runs every two minutes). Therefore, we are interested to utilize RAPTOR with strict domination to compute comprehensive instruction sets that encode these alternative routes in an easy-to-understand (for the user) and compact way.

Multimodal Journey Planning. For multicriteria multimodal journey planning, we see our approach (MCR) as a starting point rather than a final solution. Though we observed that fuzzy logic helps to rank journeys according to their significance, their rank is determined *relative* to the other journeys of the Pareto set. Therefore, we are greatly interested in an *absolute* ranking scheme that does not depend on context. While for a human it is (to some extent) relatively easy to assess the quality of a journey by just looking at it, we were yet unable to formalize a notion of absolute journey quality. An absolute ranking criterion would make it much easier to compare solutions of different algorithms, and also enable us to evaluate algorithms that compute sets of journeys by other means than by filtering (exact) Pareto sets.

Another line of work is the development of faster speedup techniques for the multimodal problem. Although, computing label-constrained shortest paths is quite efficient with UCCH, this is not yet the case for the multicriteria problem. Even with our heuristics, queries take seconds for the scenario that optimizes all modes of transport. Yet, we believe that computing concise and diverse *sets* of journeys (rather than single label-constrained journeys) is important, especially in a multimodal scenario. We are, therefore, interested in new algorithms that compute sets of alternative multimodal journeys with high quality. A first step could be to incorporate label-constraints into MCR. By these means, we could restrict the number of Pareto-optimal solutions during computation, making the algorithm faster, while still obtaining a diverse set of solutions.

Customizable Route Planning in Road Networks. Regarding our Customizable Route Planning (CRP) approach, it would be interesting to see, if the algorithm can be augmented to time-dependent route planning [DW09b] in a way such that customization is still fast. Here, a known problem is that the complexity of the associated travel time functions increases significantly for long shortcuts (i. e., clique arcs). A possible approach may approximate these travel time functions for higher levels to reduce customization space and use exact functions only for lower levels. Other scenarios, such as optimizing multiple criteria at once, are also interesting. Besides that, we believe that CRP should also be a good candidate for an external memory setting. Here, one is usually interested to minimize IOs, i. e., block reads (and writes) from (to) external memory. An interesting approach would be to minimize IOs by dropping the label-setting property of the query algorithm in order to process entire matrices at once in an apt way.

Computation of Jogging Routes. Regarding the computation of jogging routes, this thesis presented first algorithmic approaches to the problem. We, therefore, see this line of work at the beginning. Although we proved that the problem is NP-hard, we are, nevertheless, interested in comparing our algorithms to optimal solutions. Such solutions could be obtained by, e. g., branch-and-bound or linear programming approaches. Moreover, we would like to further refine the considered soft criteria, which we defined somewhat arbitrarily up to now. For that, conducting an extensive user study could be an approach. Finally, it would be interesting whether speedup techniques from road networks can be adapted to the computation of jogging routes. We believe that a good starting point is the Partial Shortest Path algorithm, which is already based on shortest paths. However, this would require the adaption of speedup techniques to compute rings (or isochrones [GSSV12, DGPW14]) instead of point-to-point shortest paths.

Algorithm Engineering. On a more general avenue, we are interested to apply the paradigm of Algorithm Engineering to other combinatorial optimization problems beyond shortest paths. Another exemplary success story of Algorithm Engineering is the graph partitioning problem [BMSW13], where many efficient algorithms were developed over the past years, some of which also benefit the computation of shortest paths. We generally believe that Algorithm Engineering can be a viable scientific principle to systematically and effectively obtain algorithmic solutions to real world problems that are both efficient and practical.

Bibliography

[ADF+11] Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck, *VC-dimension and shortest path algorithms*, Proceedings of the 38th International Colloquium on Automata, Languages, and Programming (ICALP'11), Lecture Notes in Computer Science, vol. 6755, Springer, 2011, pp. 690–699.
(Cited on pages 25 and 134.)

[ADF+12] Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck, *HLDB: Location-based services in databases*, Proceedings of the 20th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems (GIS'12), ACM Press, 2012, Best Paper Award, pp. 339–348.
(Cited on page 20.)

[ADGW11] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck, *A hub-based labeling algorithm for shortest paths on road networks*, Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11), Lecture Notes in Computer Science, vol. 6630, Springer, 2011, pp. 230–241.
(Cited on pages 20, 23, 85, 175, and 176.)

[ADGW12] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck, *Hierarchical hub labelings for shortest paths*, Proceedings of the 20th Annual European Symposium on Algorithms (ESA'12), Lecture Notes in Computer Science, vol. 7501, Springer, 2012, pp. 24–35.
(Cited on pages 15, 20, and 23.)

[ADGW13] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck, *Alternative routes in road networks*, ACM Journal of Experimental Algorithmics **18** (2013), no. 1, pp. 1–17.
(Cited on page 15.)

[AFGW10] Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck, *Highway dimension, shortest paths, and provably efficient algorithms*, Proceedings of the 21st Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'10), SIAM, 2010, pp. 782–793.
(Cited on pages 19, 24, 25, and 134.)

[AKL13] Franz Aurenhammer, Rolf Klein, and D.T. Lee, *Voronoi diagrams and delaunay triangulations*, World Scientific Publishing, August 2013.
(Cited on page 19.)

[ALS13] Julian Arz, Dennis Luxen, and Peter Sanders, *Transit node routing reconsidered*, Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13), Lecture Notes in Computer Science, vol. 7933, Springer, 2013, pp. 55–66.
(Cited on page 19.)

[Amd67] Gene M. Amdahl, *Validity of the single processor approach to achieving large scale computing capabilities*, Proceedings of the April 18-20, 1967, spring joint computer conference (New York, NY, USA), AFIPS '67 (Spring), ACM, 1967, pp. 483–485.
(Cited on page 112.)

[AW12] Leonid Antsfeld and Toby Walsh, *Finding multi-criteria optimal paths in multi-modal public transportation networks using the transit algorithm*, Proceedings of the 19th ITS World Congress, 2012.
(Cited on pages 32 and 35.)

[AZC07] Georgia Aifadopoulou, Athanasios Ziliaskopoulos, and Evangelia Chrisohoou, *Multiobjective optimum path algorithm for passenger pretrip planning in multimodal transportation networks*, Journal of the Transportation Research Board **2032** (2007), no. 1, pp. 26–34, 10.3141/2032-04.
(Cited on page 35.)

[Bas09] Hannah Bast, *Car or public transport – two worlds*, Efficient Algorithms, Lecture Notes in Computer Science, vol. 5760, Springer, 2009, pp. 355–367.
(Cited on page 30.)

[Bau12] Andreas Bauer, *Multimodal profile queries*, Bachelor thesis, Karlsruhe Institute of Technology, May 2012.
(Cited on pages 29 and 71.)

[BBH+08] Chris Barrett, Keith Bisset, Martin Holzer, Goran Konjevod, Madhav V. Marathe, and Dorothea Wagner, *Engineering label-constrained shortest-path*

algorithms, Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management (AAIM'08), Lecture Notes in Computer Science, vol. 5034, Springer, June 2008, pp. 27–37.
(Cited on page 35.)

[BBH+09] Chris Barrett, Keith Bisset, Martin Holzer, Goran Konjevod, Madhav V. Marathe, and Dorothea Wagner, *Engineering label-constrained shortest-path algorithms*, The Shortest Path Problem: Ninth DIMACS Implementation Challenge, DIMACS Book, vol. 74, American Mathematical Society, 2009, pp. 309–319.
(Cited on pages 36, 128, and 132.)

[BBJ+02] Chris Barrett, Keith Bisset, Riko Jacob, Goran Konjevod, and Madhav V. Marathe, *Classical and contemporary shortest path problems in road networks: Implementation and experimental analysis of the TRANSIMS router*, Proceedings of the 10th Annual European Symposium on Algorithms (ESA'02), Lecture Notes in Computer Science, vol. 2461, Springer, 2002, pp. 126–138.
(Cited on pages 35, 36, 128, and 131.)

[BBM06] Maurizio Bielli, Azedine Boulmakoul, and Hicham Mouncif, *Object modeling and path computation for multimodal travel systems*, European Journal of Operational Research **175** (2006), no. 3, pp. 1705–1730.
(Cited on page 35.)

[BBRW13] Reinhard Bauer, Moritz Baum, Ignaz Rutter, and Dorothea Wagner, *On the complexity of partitioning graphs for arc-flags*, Journal of Graph Algorithms and Applications **17** (2013), no. 3, pp. 265–299.
(Cited on page 24.)

[BBS13] Hannah Bast, Mirko Brodesser, and Sabine Storandt, *Result diversity for multi-modal route planning*, Proceedings of the 13th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'13), OpenAccess Series in Informatics (OASIcs), September 2013, pp. 123–136.
(Cited on pages 35 and 170.)

[BCD+08] Francesco Bruera, Serafino Cicerone, Gianlorenzo D'Angelo, Gabriele Di Stefano, and Daniele Frigioni, *Dynamic multi-level overlay graphs for shortest paths*, Mathematics in Computer Science **1** (2008), no. 4, pp. 709–736.
(Cited on page 25.)

[BCE+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, Proceedings of the 18th Annual European Symposium on Algorithms (ESA'10), Lecture Notes in Computer Science, vol. 6346, Springer, 2010, pp. 290–301.
(Cited on pages 26, 32, and 162.)

[BCK+10] Reinhard Bauer, Tobias Columbus, Bastian Katz, Marcus Krug, and Dorothea Wagner, *Preprocessing speed-up techniques is hard*, Proceedings of the 7th Conference on Algorithms and Complexity (CIAC'10), Lecture Notes in Computer Science, vol. 6078, Springer, 2010, pp. 359–370.
(Cited on page 24.)

[BCRW13] Reinhard Bauer, Tobias Columbus, Ignaz Rutter, and Dorothea Wagner, *Search-space size in contraction hierarchies*, Proceedings of the 40th International Colloquium on Automata, Languages, and Programming (ICALP'13), Lecture Notes in Computer Science, vol. 7965, Springer, 2013, pp. 93–104.
(Cited on pages 25 and 134.)

[BD09] Reinhard Bauer and Daniel Delling, *SHARC: Fast and robust unidirectional routing*, ACM Journal of Experimental Algorithmics **14** (2009), no. 2.4, pp. 1–29, Special Section on Selected Papers from ALENEX 2008.
(Cited on pages 22, 24, 31, 175, and 176.)

[BDD+12] Reinhard Bauer, Gianlorenzo D'Angelo, Daniel Delling, Andrea Schumm, and Dorothea Wagner, *The shortcut problem – complexity and algorithms*, Journal of Graph Algorithms and Applications **16** (2012), no. 2, pp. 447–481.
(Cited on page 24.)

[BDGM09] Annabell Berger, Daniel Delling, Andreas Gebhardt, and Matthias Müller–Hannemann, *Accelerating time-dependent multi-criteria timetable information is harder than expected*, Proceedings of the 9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'09), OpenAccess Series in Informatics (OASIcs), 2009.
(Cited on pages 31, 56, 109, 137, and 158.)

[BDGW10] Edith Brunel, Daniel Delling, Andreas Gemsa, and Dorothea Wagner, *Space-efficient sharc-routing*, Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10), Lecture Notes in Computer Science, vol. 6049, Springer, May 2010, pp. 47–58.
(Cited on pages 13 and 22.)

[BDPW13] Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner, *Energy-optimal routes for electric vehicles*, Tech. Report 2013-06, Faculty

of Informatics, Karlsruhe Institute of Technology, 2013.
(Cited on page 17.)

[BDS+10] 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** (2010), no. 2.3, pp. 1–31, Special Section devoted to WEA'08.
(Cited on pages 13, 20, 21, 22, 23, 24, 30, 37, 132, 175, 180, 181, and 182.)

[BDW11] Reinhard Bauer, Daniel Delling, and Dorothea Wagner, *Experimental study on speed-up techniques for timetable information systems*, Networks **57** (2011), no. 1, pp. 38–52.
(Cited on pages 21, 30, and 62.)

[Bel58] Richard Bellman, *On a routing problem*, Quarterly of Applied Mathematics **16** (1958), pp. 87–90.
(Cited on page 11.)

[BFM+07] Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes, *In transit to constant shortest-path queries in road networks*, Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07), SIAM, 2007, pp. 46–59.
(Cited on pages 18, 19, 32, 36, 132, and 175.)

[BFM09] Holger Bast, Stefan Funke, and Domagoj Matijevic, *Ultrafast shortest-path queries via transit nodes*, The Shortest Path Problem: Ninth DIMACS Implementation Challenge, DIMACS Book, vol. 74, American Mathematical Society, 2009, pp. 175–192.
(Cited on pages 18, 19, 32, and 36.)

[BFSS07] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes, *Fast routing in road networks with transit nodes*, Science **316** (2007), no. 5824, p. 566.
(Cited on pages 18, 32, and 36.)

[BGGN13] Maxim Babenko, Andrew V. Goldberg, Anupam Gupta, and Viswanath Nagarajan, *Algorithms for hub label optimization*, Proceedings of the 40th International Colloquium on Automata, Languages, and Programming (ICALP'13), Lecture Notes in Computer Science, vol. 7965, Springer, 2013, pp. 69–80.
(Cited on page 25.)

[BGM10] Annabell Berger, Martin Grimmer, and Matthias Müller–Hannemann, *Fully dynamic speed-up techniques for multi-criteria shortest path searches in*

time-dependent networks, Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10), Lecture Notes in Computer Science, vol. 6049, Springer, May 2010, pp. 35–46.
(Cited on pages 31, 56, and 98.)

[BGMO11] Annabell Berger, Andreas Gebhardt, Matthias Müller–Hannemann, and Martin Ostrowski, *Stochastic delay prediction in large train networks*, Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'11), OpenAccess Series in Informatics (OASIcs), vol. 20, 2011, pp. 100–111.
(Cited on page 33.)

[BGNS10] Gernot Veit Batz, Robert Geisberger, Sabine Neubauer, and Peter Sanders, *Time-dependent contraction hierarchies and approximation*, Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10), Lecture Notes in Computer Science, vol. 6049, Springer, May 2010, pp. 166–177.
(Cited on page 138.)

[BGSV13] Gernot Veit Batz, Robert Geisberger, Peter Sanders, and Christian Vetter, *Minimum time-dependent travel times with contraction hierarchies*, ACM Journal of Experimental Algorithmics **18** (2013), no. 1.4, pp. 1–43.
(Cited on page 15.)

[BH13] Adi Botea and Daniel Harabor, *Path planning with compressed all-pairs shortest paths data*, Proceedings of the 23rd International Conference on Automated Planning and Scheduling, AAAI Press, 2013.
(Cited on page 20.)

[BJ04] Gerth Brodal and Riko Jacob, *Time-dependent networks as models to achieve fast exact time-table queries*, Proceedings of the 3rd Workshop on Algorithmic Methods and Models for Optimization of Railways (ATMOS'03), Electronic Notes in Theoretical Computer Science, vol. 92, 2004, pp. 3–15.
(Cited on pages 27, 29, 55, 61, 65, 71, and 127.)

[BJM00] Chris Barrett, Riko Jacob, and Madhav V. Marathe, *Formal-language-constrained path problems*, SIAM Journal on Computing **30** (2000), no. 3, pp. 809–837.
(Cited on pages 4, 35, 36, 120, 125, 128, and 129.)

[BKK+07] Kevin Buchin, Christian Knauer, Klaus Kriegel, André Schulz, and Raimund Seidel, *On the number of cycles in planar graphs*, Proc. 13th International Computing and Combinatorics Conference (COCOON), 2007, pp. 97–107.
(Cited on page 193.)

[BMSW13] David A. Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner, *Graph partitioning and graph clustering: 10th dimacs implementation challenge*, vol. 588, American Mathematical Society, 2013.
(Cited on page 222.)

[Bot11] Adi Botea, *Ultra-fast optimal pathfinding without runtime search*, Proceedings of the Seventh AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE'11), AAAI Press, 2011, pp. 122–127.
(Cited on page 20.)

[BS88] Norbert Baumann and Richard Schmidt, *Buxtehude–Garmisch in 6 Sekunden. Die elektronische Fahrplanauskunft (EFA) der Deutschen Bundesbahn*, Zeitschrift für aktuelle Verkehrsfragen **10** (1988), pp. 929–931.
(Cited on page 26.)

[BSWW01] Ulrik Brandes, Frank Schulz, Dorothea Wagner, and Thomas Willhalm, *Travel planning with self-made maps*, Proceedings of the 3rd International Workshop on Algorithm Engineering and Experiments (ALENEX'01), Lecture Notes in Computer Science, vol. 2153, Springer, 2001, pp. 132–144.
(Cited on page 13.)

[CH66] K. Cooke and E. Halsey, *The shortest route through a network with time-dependent intermodal transit times*, Journal of Mathematical Analysis and Applications **14** (1966), no. 3, pp. 493–498.
(Cited on pages 28, 36, 65, 67, 126, and 129.)

[CHKZ03] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick, *Reachability and distance queries via 2-hop labels*, SIAM Journal on Computing **32** (2003), no. 5, pp. 1338–1355.
(Cited on page 25.)

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to algorithms*, 2nd ed., MIT Press, 2001.
(Cited on pages 11, 39, and 66.)

[Dan62] George B. Dantzig, *Linear programming and extensions*, Princeton University Press, 1962.
(Cited on pages 11, 36, 131, and 204.)

[DDP+12] Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato F. Werneck, *Computing and evaluating multimodal journeys*, Tech. Report 2012-20, Faculty of Informatics, Karlsruhe Institute of Technology, 2012.
(Cited on page 151.)

[DDP+13] Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato F. Werneck, *Computing multimodal journeys in practice*, Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13), Lecture Notes in Computer Science, vol. 7933, Springer, 2013, pp. 260–271.
(Cited on page 151.)

[Dea99] Brian C. Dean, *Continuous-time dynamic shortest path algorithms*, Master's thesis, Massachusetts Institute of Technology, 1999.
(Cited on pages 28, 65, 68, and 126.)

[Del09] Daniel Delling, *Engineering and augmenting route planning algorithms*, Ph.D. thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2009.
(Cited on pages 13 and 143.)

[Del11] Daniel Delling, *Time-dependent SHARC-routing*, Algorithmica **60** (2011), no. 1, pp. 60–94.
(Cited on pages 22, 30, 31, 56, and 94.)

[DF79] Eric V. Denardo and Bennett L. Fox, *Shortest-route methods: 1. Reaching, pruning, and buckets*, Operations Research **27** (1979), no. 1, pp. 161–186.
(Cited on pages 11 and 66.)

[DGJ09] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson (eds.), *The shortest path problem: Ninth dimacs implementation challenge*, DIMACS Book, vol. 74, American Mathematical Society, 2009.
(Cited on pages 10, 177, and 189.)

[DGNW13] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck, *PHAST: Hardware-accelerated shortest path trees*, Journal of Parallel and Distributed Computing **73** (2013), no. 7, pp. 940–952.
(Cited on pages 13, 15, 18, 23, and 139.)

[DGPW11] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck, *Customizable route planning*, Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11), Lecture Notes in Computer Science, vol. 6630, Springer, 2011, pp. 376–387.
(Cited on pages 15, 17, 122, 174, and 220.)

[DGPW14] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck, *Customizable route planning in road networks*, Transportation Science (2014), accpeted for publication.
(Cited on pages 17, 23, 174, and 221.)

[DGRW11] Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck, *Graph partitioning with natural cuts*, 25th International Parallel and Distributed Processing Symposium (IPDPS'11), IEEE Computer Society, 2011, pp. 1135–1146.
(Cited on pages 15, 17, and 177.)

[DGW11] Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck, *Faster batched shortest paths in road networks*, Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'11), OpenAccess Series in Informatics (OASIcs), vol. 20, 2011, pp. 52–63.
(Cited on page 15.)

[DGW13] Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck, *Hub label compression*, Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13), Lecture Notes in Computer Science, vol. 7933, Springer, 2013, pp. 18–29.
(Cited on pages 20 and 23.)

[DHM+09] Daniel Delling, Martin Holzer, Kirill Müller, Frank Schulz, and Dorothea Wagner, *High-performance multi-level routing*, The Shortest Path Problem: Ninth DIMACS Implementation Challenge, DIMACS Book, vol. 74, American Mathematical Society, 2009, pp. 73–92.
(Cited on pages 16, 19, 176, and 178.)

[Dij59] Edsger W. Dijkstra, *A note on two problems in connexion with graphs*, Numerische Mathematik **1** (1959), pp. 269–271.
(Cited on pages 9, 10, 58, 65, 66, 126, 158, 173, 176, 199, and 201.)

[DKLW12] Daniel Delling, Moritz Kobitzsch, Dennis Luxen, and Renato F. Werneck, *Robust mobile route planning with limited connectivity*, Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12), SIAM, 2012, pp. 150–159.
(Cited on page 15.)

[DKP09] Daniel Delling, Bastian Katz, and Thomas Pajor, *Parallel computation of best connections in public transportation networks*, Tech. Report 2009-16, Faculty of Informatics, Karlsruhe Institute of Technology, 2009.
(Cited on page 72.)

[DKP10] Daniel Delling, Bastian Katz, and Thomas Pajor, *Parallel computation of best connections in public transportation networks*, 24th International Parallel and Distributed Processing Symposium (IPDPS'10), IEEE Computer Society, 2010, pp. 1–12.
(Cited on page 72.)

[DKP12] Daniel Delling, Bastian Katz, and Thomas Pajor, *Parallel computation of best connections in public transportation networks*, ACM Journal of Experimental Algorithmics **17** (2012), no. 4, pp. 4.1–4.26.
(Cited on pages 55, 72, 109, and 158.)

[DMS08] Yann Disser, Matthias Müller–Hannemann, and Mathias Schnee, *Multi-criteria shortest paths in time-dependent train networks*, Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08), Lecture Notes in Computer Science, vol. 5038, Springer, June 2008, pp. 347–361.
(Cited on pages 29, 30, 33, 65, 70, 105, and 127.)

[DN08] Daniel Delling and Giacomo Nannicini, *Bidirectional core-based routing in dynamic time-dependent road networks*, Proceedings of the 19th International Symposium on Algorithms and Computation (ISAAC'08), Lecture Notes in Computer Science, vol. 5369, Springer, December 2008, pp. 813–824.
(Cited on page 138.)

[DN12] Daniel Delling and Giacomo Nannicini, *Core routing on dynamic time-dependent road networks*, Informs Journal on Computing **24** (2012), no. 2, pp. 187–201.
(Cited on pages 21, 22, and 37.)

[DP84] Narsingh Deo and Chi-Yin Pang, *Shortest-path algorithms: Taxonomy and annotation*, Networks **14** (1984), no. 2, pp. 275–323.
(Cited on page 11.)

[DPSW13] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner, *Intriguingly simple and fast transit routing*, Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13), Lecture Notes in Computer Science, vol. 7933, Springer, 2013, pp. 43–54.
(Cited on pages 28, 29, 33, 58, 67, 117, and 220.)

[DPW09a] Daniel Delling, Thomas Pajor, and Dorothea Wagner, *Accelerating multi-modal route planning by access-nodes*, Proceedings of the 17th Annual European Symposium on Algorithms (ESA'09), Lecture Notes in Computer Science, vol. 5757, Springer, September 2009, pp. 587–598.
(Cited on pages 4, 34, 35, 36, 37, 49, 121, 128, 132, 133, 138, 140, 141, 145, 147, and 148.)

[DPW09b] Daniel Delling, Thomas Pajor, and Dorothea Wagner, *Engineering time-expanded graphs for faster timetable information*, Robust and Online Large-Scale Optimization, Lecture Notes in Computer Science, vol. 5868, Springer, 2009, pp. 182–206.
(Cited on pages 27, 30, 31, 56, and 59.)

[DPW12a] Daniel Delling, Thomas Pajor, and Renato F. Werneck, *Round-based public transit routing*, Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12), SIAM, 2012, pp. 130–140.
(Cited on page 95.)

[DPW12b] Daniel Delling, Thomas Pajor, and Renato F. Werneck, *Round-Based Public Transit Routing*, Transportation Science (2012), accepted for publication, to appear.
(Cited on page 95.)

[DPW12c] Julian Dibbelt, Thomas Pajor, and Dorothea Wagner, *User-constrained multi-modal route planning*, Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12), SIAM, 2012, pp. 118–129.
(Cited on page 133.)

[DPW12d] Julian Dibbelt, Thomas Pajor, and Dorothea Wagner, *User-Constrained Multi-Modal Route Planning*, ACM Journal of Experimental Algorithmics (2012), Under Review. Date of submission.
(Cited on page 133.)

[DPWZ09] Daniel Delling, Thomas Pajor, Dorothea Wagner, and Christos Zaroliagis, *Efficient route planning in flight networks*, Proceedings of the 9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'09), OpenAccess Series in Informatics (OASIcs), 2009.
(Cited on pages 34, 123, and 124.)

[Dre69] Stuart E. Dreyfus, *An appraisal of some shortest-path algorithms*, Operations Research **17** (1969), no. 3, pp. 395–412.
(Cited on page 28.)

[DSSW09a] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner, *Engineering route planning algorithms*, Algorithmics of Large and Complex Networks, Lecture Notes in Computer Science, vol. 5515, Springer, 2009, pp. 117–139.
(Cited on pages 10, 80, 138, 183, 184, 193, and 211.)

[DSSW09b] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner, *Highway hierarchies star*, The Shortest Path Problem: Ninth DIMACS Implementation Challenge, DIMACS Book, vol. 74, American Mathematical Society, 2009, pp. 141–174.
(Cited on page 22.)

[DW07] Daniel Delling and Dorothea Wagner, *Landmark-based routing in dynamic graphs*, Proceedings of the 6th Workshop on Experimental Algorithms

(WEA'07), Lecture Notes in Computer Science, vol. 4525, Springer, June 2007, pp. 52–65.
(Cited on pages 12 and 22.)

[DW09a] Daniel Delling and Dorothea Wagner, *Pareto paths with SHARC*, Proceedings of the 8th International Symposium on Experimental Algorithms (SEA'09), Lecture Notes in Computer Science, vol. 5526, Springer, June 2009, pp. 125–136.
(Cited on pages 22 and 175.)

[DW09b] Daniel Delling and Dorothea Wagner, *Time-dependent route planning*, Robust and Online Large-Scale Optimization, Lecture Notes in Computer Science, vol. 5868, Springer, 2009, pp. 207–230.
(Cited on pages 10, 22, 34, 47, 49, 69, and 221.)

[DW13] Daniel Delling and Renato F. Werneck, *Faster customization of road networks*, Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13), Lecture Notes in Computer Science, vol. 7933, Springer, 2013, pp. 30–42.
(Cited on page 17.)

[EF12] Jochen Eisner and Stefan Funke, *Transit nodes – Lower bounds and refined construction*, Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12), SIAM, 2012, pp. 141–149.
(Cited on page 25.)

[EFS11] Jochen Eisner, Stefan Funke, and Sabine Storandt, *Optimal route planning for electric vehicles in large network*, Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence, AAAI Press, August 2011.
(Cited on page 15.)

[EG02] Matthias Ehrgott and Xavier Gandibleux (eds.), *Multiple criteria optimization: State of the art annotated bibliographic surveys*, Kluwer Academic Publishers Group, 2002.
(Cited on page 29.)

[EG08] David Eppstein and Michael T. Goodrich, *Studying (non-planar) road networks through an algorithmic lens*, Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems (GIS '08), ACM Press, 2008, pp. 1–10.
(Cited on pages 15 and 18.)

[EL11] Andrew Ensor and Felipe Lillo, *Partial order approach to compute shortest paths in multimodal networks*, Tech. report,

http://arxiv.org/abs/1112.3366v1, 2011.
(Cited on pages 35 and 121.)

[FA04] Marco Farina and Paolo Amato, *A fuzzy definition of "optimality" for many-criteria optimization problems*, IEEE Transactions on Systems, Man, and Cybernetics, Part A **34** (2004), no. 3, pp. 315–326.
(Cited on pages 121, 150, and 154.)

[FEMPS13] Eli Fox-Epstein, Shay Mozes, Phitchaya Mangpo Phothilimthana, and Christian Sommer, *Short and simple cycle separators in planar graphs*, Proceedings of the 15th Meeting on Algorithm Engineering and Experiments (ALENEX'13), SIAM, 2013, pp. 26–40.
(Cited on page 18.)

[Fli04] Ingrid C.M. Flinsenberg, *Route planning algorithms for car navigation*, Ph.D. thesis, Technische Universiteit Eindhoven, 2004.
(Cited on page 30.)

[Flo62] Robert W. Floyd, *Algorithm 97: Shortest path*, Communications of the ACM **5** (1962), no. 6, p. 345.
(Cited on page 11.)

[FMS08] Lennart Frede, Matthias Müller–Hannemann, and Mathias Schnee, *Efficient on-trip timetable information in the presence of delays*, Proceedings of the 8th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'08), OpenAccess Series in Informatics (OASIcs), September 2008.
(Cited on page 88.)

[For56] Lester R. Ford, Jr., *Network flow theory*, Tech. Report P-923, Rand Corporation, Santa Monica, California, 1956.
(Cited on page 11.)

[FS13] Stefan Funke and Sabine Storandt, *Polynomial-time construction of contraction hierarchies for multi-criteria objectives*, Proceedings of the 15th Meeting on Algorithm Engineering and Experiments (ALENEX'13), SIAM, 2013, pp. 31–54.
(Cited on page 15.)

[FSR06] L. Fu, D. Sun, and L. R. Rilett, *Heuristic shortest path algorithms for transportation applications: State of the art*, Computers & Operations Research **33** (2006), no. 11, pp. 3324–3343.
(Cited on page 10.)

[FT87] Michael L. Fredman and Robert E. Tarjan, *Fibonacci heaps and their uses in improved network optimization algorithms*, Journal of the ACM **34** (1987), no. 3, pp. 596–615.
(Cited on pages 11 and 66.)

[Gei10] Robert Geisberger, *Contraction of timetable networks with realistic transfers*, Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10), Lecture Notes in Computer Science, vol. 6049, Springer, May 2010, pp. 71–82.
(Cited on pages 31, 109, 135, 137, and 158.)

[Gen10] General Transit Feed, `https://developers.google.com/transit/gtfs/`, 2010.
(Cited on pages 87, 113, 141, and 169.)

[GH05] Andrew V. Goldberg and Chris Harrelson, *Computing the shortest path: A* search meets graph theory*, Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05), SIAM, 2005, pp. 156–165.
(Cited on pages 11, 12, 21, 30, 36, 37, 131, 132, 175, and 176.)

[GJ79] Michael R. Garey and David S. Johnson, *Computers and intractability: A guide to the theory of np-completeness*, W. H. Freeman & Co., New York, NY, USA, 1979.
(Cited on page 195.)

[GKM+11] Marc Goerigk, Martin Knoth, Matthias Müller–Hannemann, Marie Schmidt, and Anita Schöbel, *The price of robustness in timetable information*, Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'11), OpenAccess Series in Informatics (OASIcs), vol. 20, 2011, pp. 76–87.
(Cited on page 33.)

[GKM+13] Marc Goerigk, Martin Knoth, Matthias Müller–Hannemann, Marie Schmidt, and Anita Schöbel, *The price of strict and light robustness in timetable information*, Transportation Science (2013), Published online before print.
(Cited on page 33.)

[GKS10] Robert Geisberger, Moritz Kobitzsch, and Peter Sanders, *Route planning with flexible objective functions*, Proceedings of the 12th Workshop on Algorithm Engineering and Experiments (ALENEX'10), SIAM, 2010, pp. 124–137.
(Cited on pages 15, 135, 175, and 183.)

[GKW07] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck, *Better landmarks within reach*, Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07), Lecture Notes in Computer Science, vol. 4525, Springer, June 2007, pp. 38–51.
(Cited on page 22.)

[GKW09] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck, *Reach for A*: Shortest path algorithms with preprocessing*, The Shortest Path Problem: Ninth DIMACS Implementation Challenge, DIMACS Book, vol. 74, American Mathematical Society, 2009, pp. 93–139.
(Cited on pages 14, 21, 175, and 181.)

[GLS+10] Robert Geisberger, Dennis Luxen, Peter Sanders, Sabine Neubauer, and Lars Volker, *Fast detour computation for ride sharing*, Proceedings of the 10th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'10), OpenAccess Series in Informatics (OASIcs), vol. 14, 2010, pp. 88–99.
(Cited on page 15.)

[GMS07] Thorsten Gunkel, Matthias Müller–Hannemann, and Mathias Schnee, *Improved search for night train connections*, Proceedings of the 7th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'07), OpenAccess Series in Informatics (OASIcs), 2007, pp. 243–258.
(Cited on page 33.)

[Gol01] Andrew V. Goldberg, *A simple shortest path algorithm with linear average time*, Proceedings of the 9th Annual European Symposium on Algorithms (ESA'01), Lecture Notes in Computer Science, vol. 2161, 2001, pp. 230–241.
(Cited on page 11.)

[Goo10] Google, *Google Transit*, `http://www.google.com`, 2010.
(Cited on pages 26 and 32.)

[GP03] Cyril Gavoille and David Peleg, *Compact and localized distributed data structures*, Distributed Computing **16** (2003), no. 2–3, pp. 111–120.
(Cited on page 18.)

[GPPR04] Cyril Gavoille, David Peleg, Stéphane Pérennes, and Ran Raz, *Distance labeling in graphs*, Journal of Algorithms **53** (2004), pp. 85–112.
(Cited on pages 19 and 25.)

[GPWZ13] Andreas Gemsa, Thomas Pajor, Dorothea Wagner, and Tobias Zündorf, *Efficient computation of jogging routes*, Proceedings of the 12th International

Symposium on Experimental Algorithms (SEA'13), Lecture Notes in Computer Science, vol. 7933, Springer, 2013, pp. 272–283.
(Cited on page 194.)

[Gra72] Ronald L. Graham, *An efficient algorithm for determining the convex hull of a finite planar set*, Information Processing Letters **1** (1972), no. 4, pp. 132–133.
(Cited on page 200.)

[GRST12] Robert Geisberger, Michael Rice, Peter Sanders, and Vassilis Tsotras, *Route planning with flexible edge restrictions*, ACM Journal of Experimental Algorithmics **17** (2012), no. 1, pp. 1–20.
(Cited on pages 15, 37, and 38.)

[GSSD08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling, *Contraction hierarchies: Faster and simpler hierarchical routing in road networks*, Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08), Lecture Notes in Computer Science, vol. 5038, Springer, June 2008, pp. 319–333.
(Cited on pages 15, 133, 143, and 175.)

[GSSV12] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter, *Exact routing in large road networks using contraction hierarchies*, Transportation Science **46** (2012), no. 3, pp. 388–404.
(Cited on pages 4, 15, 19, 21, 23, 24, 31, 36, 37, 85, 121, 131, 132, 133, 134, 136, 138, 143, 150, 160, 175, 176, 183, 190, and 221.)

[Gut04] Ronald J. Gutman, *Reach-based routing: A new approach to shortest path algorithms optimized for road networks*, Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX'04), SIAM, 2004, pp. 100–111.
(Cited on pages 14 and 176.)

[GV11] Robert Geisberger and Christian Vetter, *Efficient routing in road networks with turn costs*, Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11), Lecture Notes in Computer Science, vol. 6630, Springer, 2011, pp. 100–111.
(Cited on pages 15, 122, and 186.)

[GW05] Andrew V. Goldberg and Renato F. Werneck, *Computing point-to-point shortest paths from external memory*, Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05), SIAM, 2005, pp. 26–40.
(Cited on page 12.)

[HaC] HaCon Ingenieurgesellschaft mbH, *HAFAS – A timetable information system, Hannover, Germany*, `http://www.hacon.de/hafas`.
(Cited on pages 1 and 26.)

[HaC84] HaCon - Ingenieurgesellschaft mbH, `http://www.hacon.de`, 1984.
(Cited on pages 26, 87, and 141.)

[Han79] Pierre Hansen, *Bricriteria path problems*, Multiple Criteria Decision Making – Theory and Application –, Springer, 1979, pp. 109–127.
(Cited on pages 29, 69, and 152.)

[HJR96] Yun-Wu Huang, Ning Jing, and Elke A. Rundensteiner, *Effective graph clustering for path queries in digital maps*, Proceedings of the 5th International Conference on Information and Knowledge Management, ACM Press, 1996, pp. 215–222.
(Cited on page 176.)

[HKMS09] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling, *Fast point-to-point shortest path computations with arc-flags*, The Shortest Path Problem: Ninth DIMACS Implementation Challenge, DIMACS Book, vol. 74, American Mathematical Society, 2009, pp. 41–72.
(Cited on pages 13, 36, 131, and 175.)

[HNR68] Peter E. Hart, Nils Nilsson, and Bertram Raphael, *A formal basis for the heuristic determination of minimum cost paths*, IEEE Transactions on Systems Science and Cybernetics **4** (1968), pp. 100–107.
(Cited on pages 12, 30, 36, and 131.)

[Hol08] Martin Holzer, *Engineering planar-separator and shortest-path algorithms*, Ph.D. thesis, Karlsruhe Institute of Technology (KIT) - Department of Informatics, 2008.
(Cited on pages 36 and 132.)

[HSW04] Martin Holzer, Frank Schulz, and Thomas Willhalm, *Combining speed-up techniques for shortest-path computations*, Proceedings of the 3rd Workshop on Experimental Algorithms (WEA'04), Lecture Notes in Computer Science, vol. 3059, Springer, 2004, pp. 269–284.
(Cited on page 21.)

[HSW08] Martin Holzer, Frank Schulz, and Dorothea Wagner, *Engineering multilevel overlay graphs for shortest-path queries*, ACM Journal of Experimental Algorithmics **13** (2008), no. 2.5, pp. 1–26.
(Cited on pages 16, 31, 174, 176, 178, and 191.)

[HSWW06] Martin Holzer, Frank Schulz, Dorothea Wagner, and Thomas Willhalm, *Combining speed-up techniques for shortest-path computations*, ACM Journal of Experimental Algorithmics **10** (2006), no. 2.5, pp. 1–18.
(Cited on pages 20, 21, and 30.)

[JP02] Sungwon Jung and Sakti Pramanik, *An efficient path computation model for hierarchically structured topographical road maps*, IEEE Transactions on Knowledge and Data Engineering **14** (2002), no. 5, pp. 1029–1046.
(Cited on pages 5, 16, 17, 176, 177, 181, and 220.)

[Kar72] Richard M. Karp, *Reducibility among combinatorial problems*, Complexity of Computer Computations, Plenum Press, 1972, pp. 85–103.
(Cited on pages 63 and 194.)

[Kar07] George Karypis, *Metis - family of multilevel partitioning algorithms*, 2007.
(Cited on page 16.)

[KK97] Hermann Kaindl and Gerhard Kainz, *Bidirectional heuristic search reconsidered*, Journal of Artificial Intelligence Research **7** (1997), pp. 283–317.
(Cited on page 21.)

[KK99] George Karypis and Gautam Kumar, *A fast and highly quality multilevel scheme for partitioning irregular graphs*, SIAM Journal on Scientific Computing **20** (1999), no. 1, pp. 359–392.
(Cited on page 177.)

[KLC12] Dominik Kirchler, Leo Liberti, and Roberto Wolfler Calvo, *A label correcting algorithm for the shortest path problem on a multi-modal route network*, Proceedings of the 11th International Symposium on Experimental Algorithms (SEA'12), Lecture Notes in Computer Science, vol. 7276, Springer, 2012, pp. 236–247.
(Cited on pages 34, 35, 37, 128, and 132.)

[Kle56] Stephen Cole Kleene, *Representation of events in nerve nets and finite automata*, Automata Studies, Annals of Mathematics Studies, Princeton University Press, 1956, pp. 3–42.
(Cited on page 45.)

[KLPC11] Dominik Kirchler, Leo Liberti, Thomas Pajor, and Roberto Wolfler Calvo, *UniALT for regular language constraint shortest paths on a multi-modal transportation network*, Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'11), OpenAccess Series in Informatics (OASIcs), vol. 20, 2011, pp. 64–75.
(Cited on pages 34, 35, 37, 128, and 132.)

[KLSV10] Tim Kieritz, Dennis Luxen, Peter Sanders, and Christian Vetter, *Distributed time-dependent contraction hierarchies*, Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10), Lecture Notes in Computer Science, vol. 6049, Springer, May 2010, pp. 83–93.
(Cited on page 15.)

[KMS05] Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling, *Acceleration of shortest path and constrained shortest path computation*, Proceedings of the 4th Workshop on Experimental Algorithms (WEA'05), Lecture Notes in Computer Science, vol. 3503, Springer, 2005, pp. 126–138.
(Cited on page 13.)

[KSS+07] Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner, *Computing many-to-many shortest paths using highway hierarchies*, Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07), SIAM, 2007, pp. 36–45.
(Cited on page 15.)

[Lau97] Ulrich Lauther, *Slow preprocessing of graphs for extremely fast shortest path calculations*, 1997, Lecture at the Workshop on Computational Integer Programming at ZIB.
(Cited on page 13.)

[Lau04] Ulrich Lauther, *An extremely fast, exact algorithm for finding shortest paths in static networks with geographical background*, Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung, vol. 22, IfGI prints, 2004, pp. 219–230.
(Cited on pages 13, 31, and 176.)

[Lau09] Ulrich Lauther, *An experimental evaluation of point-to-point shortest path calculation on roadnetworks with precalculated edge-flags*, The Shortest Path Problem: Ninth DIMACS Implementation Challenge, DIMACS Book, vol. 74, American Mathematical Society, 2009, pp. 19–40.
(Cited on pages 36 and 131.)

[Lon11] London Data Store, `http://data.london.gov.uk/`, 2011.
(Cited on pages 107 and 163.)

[Lor84] P Loridan, *ϵ-solutions in vector minimization problems*, Journal of Optimization Theory and Applications **43** (1984), no. 2, pp. 265–276.
(Cited on page 29.)

[LS12] Dennis Luxen and Dennis Schieferdecker, *Candidate sets for alternative routes in road networks*, Proceedings of the 11th International Symposium

on Experimental Algorithms (SEA'12), Lecture Notes in Computer Science, vol. 7276, Springer, 2012, pp. 260–270.
(Cited on page 15.)

[LT79] Richard J. Lipton and Robert E. Tarjan, *A separator theorem for planar graphs*, SIAM Journal on Applied Mathematics **36** (1979), no. 2, pp. 177–189.
(Cited on pages 16, 17, and 177.)

[LV11] Dennis Luxen and Christian Vetter, *Real-time routing with OpenStreetMap data*, Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems, ACM Press, 2011.
(Cited on page 15.)

[LW06] Hongbo Liu and Jiaxin Wang, *A new way to enumerate cycles in graph*, Proceedings of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services, AICT-ICIW '06, IEEE Computer Society, 2006, pp. 57–60.
(Cited on page 193.)

[Mac67] J. MacQueen, *Some methods for classification and analysis of multivariate observations*, Fifth Berkeley Symposium on Mathematical Statistics and Probability, 1967, pp. 281–297.
(Cited on page 78.)

[Mar84] Ernesto Queiros Martins, *On a multicriteria shortest path problem*, European Journal of Operational Research **26** (1984), no. 3, pp. 236–245.
(Cited on page 29.)

[MBBC09] Kamesh Madduri, David A. Bader, Jonathan W. Berry, and Joseph R. Crobak, *Parallel shortest path algorithms for solving large-scale instances*, The Shortest Path Problem: Ninth DIMACS Implementation Challenge, DIMACS Book, vol. 74, American Mathematical Society, 2009, pp. 249–290.
(Cited on pages 49 and 101.)

[Men] Mentz Datenverarbeitung GmbH, *EFA – A timetable information system, München, Germany*, `http://www.mentzdv.de`.
(Cited on page 26.)

[Met66] Metropolitan Transportation Authority of the State of New York, `http://www.mta.info/`, 1966.
(Cited on page 140.)

[Mey01] Ulrich Meyer, *Single-source shortest-paths on arbitrary directed graphs in linear average-case time*, Proceedings of the 12th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'01), 2001, pp. 797–806.
(Cited on page 11.)

[Mic12] Microsoft, *Bing maps new routing engine*, January 2012, `http://www.bing.com/blogs/site_blogs/b/maps/archive/2012/01/05/bing-maps-new-routing-engine.aspx`.
(Cited on pages 5, 17, and 191.)

[Mil86] Gary L. Miller, *Finding small simple cycle separators for 2–connected planar graphs*, Journal of Computer and System Sciences **32** (1986), no. 3, pp. 265–279.
(Cited on page 18.)

[Möh99] Rolf H. Möhring, *Verteilte Verbindungssuche im öffentlichen Personenverkehr – Graphentheoretische Modelle und Algorithmen*, Angewandte Mathematik insbesondere Informatik, Beispiele erfolgreicher Wege zwischen Mathematik und Informatik, Vieweg, 1999, pp. 192–220.
(Cited on pages 26 and 29.)

[Mor92] H. Moritz, *Geodetic reference system 1980*, Journal of Geodesy **66** (1992), no. 2, pp. 187–192.
(Cited on page 125.)

[MS98] Paola Modesti and Anna Sciomachen, *A utility measure for finding multiobjective shortest paths in urban multimodal transportation networks*, European Journal of Operational Research **111** (1998), no. 3, pp. 495–508.
(Cited on page 35.)

[MS03] Ulrich Meyer and Peter Sanders, *δ-stepping: A parallelizable shortest path algorithm*, Journal of Algorithms **49** (2003), no. 1, pp. 114–152.
(Cited on pages 49 and 101.)

[MS06] Matthias Müller–Hannemann and Mathias Schnee, *Paying less for train connections with motis*, Proceedings of the 5th Workshop on Algorithmic Methods and Models for Optimization of Railways (ATMOS'05), OpenAccess Series in Informatics (OASIcs), 2006, p. 657.
(Cited on page 33.)

[MS07] Matthias Müller–Hannemann and Mathias Schnee, *Finding all attractive train connections by multi-criteria pareto search*, Algorithmic Methods for Railway Optimization, Lecture Notes in Computer Science, vol. 4359, Springer, 2007, pp. 246–263.
(Cited on pages 28 and 29.)

[MS09] Matthias Müller–Hannemann and Mathias Schnee, *Efficient timetable information in the presence of delays*, Robust and Online Large-Scale Optimization, Lecture Notes in Computer Science, vol. 5868, Springer, 2009, pp. 249–272.
(Cited on page 32.)

[MS10] Matthias Müller–Hannemann and Stefan Schirra (eds.), *Algorithm engineering: Bridging the gap between algorithm theory and practice*, Lecture Notes in Computer Science, vol. 5971, Springer, 2010.
(Cited on pages v, 2, and 264.)

[MSM09] Jens Maue, Peter Sanders, and Domagoj Matijevic, *Goal-directed shortest-path queries using precomputed cluster distances*, ACM Journal of Experimental Algorithmics **14** (2009), pp. 3.2:1–3.2:27.
(Cited on pages 13, 175, 176, and 180.)

[MSS+06] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm, *Partitioning graphs to speedup Dijkstra's algorithm*, ACM Journal of Experimental Algorithmics **11** (2006), no. 2.8, pp. 1–29.
(Cited on page 13.)

[MSWZ07] Matthias Müller–Hannemann, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis, *Timetable information: Models and algorithms*, Algorithmic Methods for Railway Optimization, Lecture Notes in Computer Science, vol. 4359, Springer, 2007, pp. 67–90.
(Cited on pages 26, 150, and 158.)

[Mül06] Kirill Müller, *Design and implementation of an efficient hierarchical speed-up technique for computation of exact shortest paths in graphs*, Master's thesis, Universität Karlsruhe (TH), Fakultät für Informatik, June 2006.
(Cited on page 19.)

[MW95] Alberto O. Mendelzon and Peter T. Wood, *Finding regular simple paths in graph databases*, SIAM Journal on Computing **24** (1995), no. 6, pp. 1235–1258.
(Cited on page 35.)

[MW01] Matthias Müller–Hannemann and Karsten Weihe, *Pareto shortest paths is often feasible in practice*, Proceedings of the 5th International Workshop on Algorithm Engineering (WAE'01), Lecture Notes in Computer Science, vol. 2141, Springer, 2001, pp. 185–197.
(Cited on pages 26, 28, and 152.)

[MZ07] Laurent Flindt Muller and Martin Zachariasen, *Fast and compact oracles for approximate distances in planar graphs*, Proceedings of the 14th Annual

European Symposium on Algorithms (ESA'07), Lecture Notes in Computer Science, vol. 4698, Springer, 2007, pp. 657–668.
(Cited on page 176.)

[Nac95] Karl Nachtigall, *Time depending shortest-path problems with applications to railway networks*, European Journal of Operational Research **83** (1995), no. 1, pp. 154–166.
(Cited on pages 27, 28, and 55.)

[Ope04] OpenStreetMap, `http://openstreetmap.org/`, 2004.
(Cited on pages 15 and 205.)

[Ope12] OpenTripPlanner, *OpenTripPlanner,* `http://opentripplanner.com`, 2012.
(Cited on page 26.)

[OR90] Ariel Orda and Raphael Rom, *Shortest-path and minimum delay algorithms in networks with time-dependent edge-length*, Journal of the ACM **37** (1990), no. 3, pp. 607–625.
(Cited on pages 27 and 28.)

[OR91] Ariel Orda and Raphael Rom, *Minimum weight paths in time-dependent networks*, Networks **21** (1991), pp. 295–319.
(Cited on pages 27 and 28.)

[Paj09] Thomas Pajor, *Multi-modal route planning*, Master's thesis, Universität Karlsruhe (TH), March 2009.
(Cited on pages 34, 35, 36, 123, 128, 129, and 131.)

[Par61] Seymour V. Parter, *The use of linear graphs in Gauss elimination*, SIAM Review **3** (1961), no. 2, pp. 119–130.
(Cited on page 25.)

[Pel00] David Peleg, *Proximity-preserving labeling schemes*, Journal of Graph Theory **33** (2000), no. 3, pp. 167–176.
(Cited on page 19.)

[Poh69] Ira Pohl, *Bi-directional and heuristic search in path problems*, Tech. Report SLAC-104, Stanford Linear Accelerator Center, Stanford, California, 1969.
(Cited on page 21.)

[Poh71] Ira Pohl, *Bi-directional search*, Proceedings of the Sixth Annual Machine Intelligence Workshop, vol. 6, Edinburgh University Press, 1971, pp. 124–140.
(Cited on page 12.)

[Pop34] Sir Karl Popper, *The logic of scientific discovery*, Hutchinson, 1934.
(Cited on page 2.)

[PS98] Stefano Pallottino and Maria Grazia Scutellà, *Shortest path algorithms in transportation models: Classical and innovative aspects*, Equilibrium and Advanced Transportation Modelling, Kluwer Academic Publishers Group, 1998, pp. 245–281.
(Cited on page 26.)

[PSWZ04a] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis, *Experimental comparison of shortest path approaches for timetable information*, Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX'04), SIAM, 2004, pp. 88–99.
(Cited on page 27.)

[PSWZ04b] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis, *Towards realistic modeling of time-table information through the time-dependent approach*, Proceedings of the 3rd Workshop on Algorithmic Methods and Models for Optimization of Railways (ATMOS'03), Electronic Notes in Theoretical Computer Science, vol. 92, 2004, pp. 85–103.
(Cited on pages 55, 61, and 123.)

[PSWZ08] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis, *Efficient models for timetable information in public transportation systems*, ACM Journal of Experimental Algorithmics **12** (2008), no. 2.4, pp. 1–39.
(Cited on pages 3, 26, 27, 29, 34, 55, 56, 59, 61, 62, 65, 70, 123, 127, and 137.)

[PTV79] PTV AG – Planung Transport Verkehr, *http://www.ptv.de*, 1979.
(Cited on pages 141, 163, 169, and 177.)

[PY00] Christos H. Papadimitriou and Mihalis Yannakakis, *On the approximability of trade-offs and optimal access of web sources*, Proceedings of the 41st Annual IEEE Symposium on Foundations of Computer Science (FOCS'00), 2000, pp. 86–92.
(Cited on page 29.)

[RS59] Michael Oser Rabin and Dana Scott, *Finite automata and their decision problems*, IBM Journal of Research and Development **3** (1559), pp. 114–125.
(Cited on page 45.)

[RT10] Michael Rice and Vassilis Tsotras, *Graph indexing of road networks for shortest path queries with label restrictions*, Proceedings of the VLDB Endowment

4 (2010), no. 3, pp. 69–80.
(Cited on pages 35, 37, 38, 128, 129, and 175.)

[RT12] Michael Rice and Vassilis Tsotras, *Bidirectional A* search with additive approximation bounds*, Proceedings of the 5th International Symposium on Combinatorial Search (SoCS'12), AAAI Press, 2012.
(Cited on page 25.)

[San09] Peter Sanders, *Algorithm engineering – an attempt at a definition*, Efficient Algorithms, Lecture Notes in Computer Science, vol. 5760, Springer, 2009, pp. 321–340.
(Cited on pages v, 2, and 264.)

[Sas11] Jan-Ole Sasse, *Route planning in road networks with turn costs and multi edge restrictions*, Diploma thesis, Karlsruhe Institute of Technology, November 2011.
(Cited on pages 122 and 123.)

[Sch82] Robert Schreiber, *A new implementation of sparse Gaussian elimination*, ACM Transactions on Mathematical Software **8** (1982), no. 3, pp. 256–276.
(Cited on page 25.)

[Sch12] Heiko Schilling, *TomTom navigation – How mathematics help getting through traffic faster*, 2012, Talk given at ISMP.
(Cited on page 122.)

[Sen09] Sandeep Sen, *Approximating shortest paths in graphs*, Proceedings of the 3rd Workshop on Algorithms and Computation (WALCOM'09), Lecture Notes in Computer Science, vol. 5431, Springer, February 2009, pp. 32–43.
(Cited on page 18.)

[Som12] Christian Sommer, *Shortest-path queries in static networks*, 2012, submitted. Preprint available at `http://www.sommer.jp/spq-survey.htm`.
(Cited on pages 10, 18, 23, 193, and 211.)

[SS05] Peter Sanders and Dominik Schultes, *Highway hierarchies hasten exact shortest path queries*, Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05), Lecture Notes in Computer Science, vol. 3669, Springer, 2005, pp. 568–579.
(Cited on pages 15, 36, 109, 131, and 187.)

[SS07] Dominik Schultes and Peter Sanders, *Dynamic highway-node routing*, Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07),

Lecture Notes in Computer Science, vol. 4525, Springer, June 2007, pp. 66–79.
(Cited on page 15.)

[SS09] Peter Sanders and Dominik Schultes, *Robust, almost constant time shortest-path queries in road networks*, The Shortest Path Problem: Ninth DIMACS Implementation Challenge, DIMACS Book, vol. 74, American Mathematical Society, 2009, pp. 193–218.
(Cited on pages 18, 19, 32, 36, and 176.)

[SS12a] Peter Sanders and Dominik Schultes, *Engineering highway hierarchies*, ACM Journal of Experimental Algorithmics **17** (2012), no. 1, pp. 1–40.
(Cited on pages 15 and 22.)

[SS12b] Peter Sanders and Christian Schulz, *Distributed evolutionary graph partitioning*, Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12), SIAM, 2012, pp. 16–29.
(Cited on page 15.)

[SSV08] Peter Sanders, Dominik Schultes, and Christian Vetter, *Mobile route planning*, Proceedings of the 16th Annual European Symposium on Algorithms (ESA'08), Lecture Notes in Computer Science, vol. 5193, Springer, September 2008, pp. 732–743.
(Cited on page 15.)

[SV86] Robert Sedgewick and Jeffrey S. Vitter, *Shortest paths in Euclidean graphs*, Algorithmica **1** (1986), no. 1, pp. 31–48.
(Cited on pages 12, 36, and 132.)

[SW13] Peter Sanders and Dorothea Wagner, *Algorithm engineering*, Informatik Spektrum **36** (2013), no. 2, pp. 187–190.
(Cited on pages v, 2, and 264.)

[SWW99] Frank Schulz, Dorothea Wagner, and Karsten Weihe, *Dijkstra's algorithm on-line: An empirical case study from public railroad transport*, Proceedings of the 3rd International Workshop on Algorithm Engineering (WAE'99), Lecture Notes in Computer Science, vol. 1668, Springer, 1999, pp. 110–123.
(Cited on pages 9 and 26.)

[SWW00] 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** (2000), no. 12, pp. 1–23.
(Cited on pages 5, 13, 14, 16, 20, 21, 26, 27, 28, 30, 31, 36, 55, 65, 85, 117, 131, 177, 178, and 220.)

[SWZ02] Frank Schulz, Dorothea Wagner, and Christos Zaroliagis, *Using multi-level graphs for timetable information in railway systems*, Proceedings of the 4th Workshop on Algorithm Engineering and Experiments (ALENEX'02), Lecture Notes in Computer Science, vol. 2409, Springer, 2002, pp. 43–59.
(Cited on pages 16, 31, 55, 117, 176, 178, and 220.)

[The95] Dirk Theune, *Robuste und effiziente methoden zur lösung von wegproblemen*, Ph.D. thesis, Universität Paderborn, 1995.
(Cited on page 29.)

[Tra00] Transport for London, `http://www.tfl.gov.uk/`, 2000.
(Cited on pages 104, 107, 124, and 163.)

[TS88] Eduard Tulp and Laurent Siklóssy, *TRAINS, An Active Time-Table Searcher*, ECAI, vol. 88, 1988, pp. 170–175.
(Cited on page 26.)

[TS91] Eduard Tulp and Laurent Siklóssy, *Searching Time-Table Networks*, Artificial Intelligence for Engineering Design, Analysis and Manufacturing **5** (1991), no. 3, pp. 189–198.
(Cited on page 26.)

[TZ06] George Tsaggouris and Christos Zaroliagis, *Multiobjective optimization: Improved FPTAS for shortest paths and non-linear objectives with applications*, Proceedings of the 17th International Symposium on Algorithms and Computation (ISAAC'06), Lecture Notes in Computer Science, vol. 4288, Springer, 2006, pp. 389–398.
(Cited on page 29.)

[VC71] Vladimir N. Vapnik and Alexey Ya. Chervonenkis, *On the uniform convergence of relative frequencies of events to their probabilities*, Theory of Probability and its Applications **16** (1971), no. 2, pp. 264–280.
(Cited on page 25.)

[Vor08] Georges Voronoi, *Nouvelles applications des paramètres continus à la théorie des formes quadratiques. Deuxième mémoire. Recherches sur les parallélloèdres primitifs.*, Journal für die reine und angewandte Mathematik (Crelles Journal) **1908** (1908), no. 134, pp. 198–287.
(Cited on page 19.)

[Whi86] Douglas J White, *Epsilon efficiency*, Journal of Optimization Theory and Applications **49** (1986), no. 2, pp. 319–337.
(Cited on page 29.)

[WW05] Dorothea Wagner and Thomas Willhalm, *Drawing graphs to speed up shortest-path computations*, Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05), SIAM, 2005, pp. 15–24.
(Cited on page 13.)

[WW07] Dorothea Wagner and Thomas Willhalm, *Speed-up techniques for shortest-path computations*, Proceedings of the 24th International Symposium on Theoretical Aspects of Computer Science (STACS'07), Lecture Notes in Computer Science, vol. 4393, Springer, 2007, Invited Talk, pp. 23–36.
(Cited on page 10.)

[WWZ05] Dorothea Wagner, Thomas Willhalm, and Christos Zaroliagis, *Geometric containers for efficient shortest-path computation*, ACM Journal of Experimental Algorithmics **10** (2005), no. 1.3, pp. 1–30.
(Cited on pages 13, 20, and 30.)

[YL12] Haicong Yu and Feng Lu, *Advanced multi-modal routing approach for pedestrians*, 2nd International Conference on Consumer Electronics, Communications and Networks, 2012, pp. 2349–2352.
(Cited on pages 34 and 35.)

[YZ95] Raphael Yuster and Uri Zwick, *Color-coding*, Journal of the ACM **42** (1995), no. 4, pp. 844–856.
(Cited on page 193.)

[Zad65] Lotfi A. Zadeh, *Fuzzy sets*, Information and Control **8** (1965), no. 3, pp. 338–353.
(Cited on pages 150 and 152.)

[Zad88] Lotfi A. Zadeh, *Fuzzy logic*, IEEE Computer **21** (1988), no. 4, pp. 83–93.
(Cited on pages 121, 150, and 152.)

[Zün12] Tobias Zündorf, *Effiziente berechnung guter joggingrouten*, Bachelor thesis, Karlsruhe Institute of Technology, October 2012.
(Cited on page 194.)

[Zwi01] Uri Zwick, *Exact and approximate distances in graphs – A survey*, Proceedings of the 9th Annual European Symposium on Algorithms (ESA'01), Lecture Notes in Computer Science, vol. 2161, 2001, pp. 33–48.
(Cited on page 18.)

[Zü90] Züricher Verkehrsverbund, `http://www.zvv.ch`, 1990.
(Cited on page 104.)

List of Figures

List of Tables

Appendix A

Curriculum Vitæ

Thomas Pajor

born 31 October 1982 in Potsdam, Germany

Current Status

since 10/2013	Post doc researcher at Microsoft Research Silicon Valley

Education

04/2009–07/2013	PhD student in Informatics Karlsruhe Institute of Technology (KIT) Advisors: Prof. Dr. Dorothea Wagner, Prof. Dr. Matthias Müller-Hannemann
10/2003–03/2009	Diploma (German M. Sc.) with distinction in Informatics Universität Fridericiana zu Karlsruhe (TH) Thesis: Multi-Modal Route Planning
06/2002	Abitur (final secondary school examinations) Hochrhein-Gymnasium Waldshut

Experience Abroad

08/2011–10/2011	Internship at Microsoft Research Silicon Valley Supervisors: Daniel Delling and Renato F. Werneck Researched on multicriteria public transit route planning
03/2011–05/2011	Contractor for Microsoft Consulting Services UK Developed a journey planning engine for Transport for London
07/2010–10/2010	Internship at Microsoft Research Silicon Valley Supervisors: Daniel Delling, Andrew Goldberg, Renato Werneck

	Researched on customizable route planning in road networks
09/2008–01/2009	Research group of Prof. Dr. Christos Zaroliagis Department of Computer Engineering & Informatics University of Patras, Greece

Awards

07/2013	Teaching award for the best lecture in the summer term 2012 Awarded for the course "Algorithms for Route Planning"
10/2009	Graduation Award of the City of Karlsruhe 2009
07/2009	Diploma with distinction Universität Fridericiana zu Karlsruhe (TH)

Teaching Activities

04/2013–07/2013	Lecture "Algorithms for Route Planning"
10/2012–03/2013	Practical course "Algorithm Engineering"
04/2012–07/2012	Lecture "Algorithms for Route Planning"
10/2011–03/2012	Practical course "Algorithm Engineering"
04/2011–07/2011	Lecture "Algorithms for Route Planning"
04/2010–07/2010	Lecture "Algorithms for Route Planning"
10/2009–03/2010	Teaching assistant for "Algorithms and Data Structures"
05/2009–09/2009	Teaching assistant for "Algorithms for Route Planning"

Appendix B

List of Publications

All conference, journal, and book publications have been peer-reviewed.

Book Chapters and Journal Articles

Customizable route planning in road networks. *Transportation Science*, 2014, accpeted for publication. Joint work with Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck.

On d-regular schematization of embedded paths. *Computational Geometry: Theory and Applications*, 47(3A):381–406, 2014. Joint work with Daniel Delling, Andreas Gemsa, Martin Nöllenburg, and Ignaz Rutter.

Round-based public transit routing. *Transportation Science*, 2014, accepted for publication. Joint work with Daniel Delling and Renato F. Werneck.

Parallel computation of best connections in public transportation networks. *ACM Journal of Experimental Algorithmics*, 17(4):4.1–4.26, July 2012. Joint work with Daniel Delling and Bastian Katz.

Engineering time-expanded graphs for faster timetable information. In: *Robust and Online Large-Scale Optimization*, volume 5868 of *Lecture Notes in Computer Science*, pages 182–206. Springer, 2009. Joint work with Daniel Delling and Dorothea Wagner.

Conference Proceedings

Computing multimodal journeys in practice. In: *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 260–271. Springer, 2013. Joint work with Daniel Delling, Julian Dibbelt, Dorothea Wagner, and Renato F. Werneck.

Intriguingly simple and fast transit routing. In: *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 43–54. Springer, 2013. Joint work with Julian Dibbelt, Ben Strasser, and Dorothea Wagner.

Efficient computation of jogging routes. In: *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 272–283. Springer, 2013. Joint work with Andreas Gemsa, Dorothea Wagner, and Tobias Zündorf.

Round-based public transit routing. In: *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*, pages 130–140. SIAM, 2012. Joint work with Daniel Delling and Renato F. Werneck.

User-constrained multi-modal route planning. In: *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*, pages 118–129. SIAM, 2012. Joint work with Julian Dibbelt and Dorothea Wagner.

Customizable route planning. In: *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *Lecture Notes in Computer Science*, pages 376–387. Springer, 2011. Joint work with Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck.

UniALT for regular language constraint shortest paths on a multi-modal transportation network. In: *Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'11)*, volume 20 of *OpenAccess Series in Informatics (OASIcs)*, pages 64–75, 2011. Joint work with Dominik Kirchler, Leo Liberti, and Roberto Wolfler Calvo.

On d-regular schematization of embedded paths. In: *Proceedings of the 37th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'11)*, volume 6543 of *Lecture Notes in Computer Science*, pages 260–271. Springer, January 2011. Joint work with Andreas Gemsa, Martin Nöllenburg, and Ignaz Rutter.

Automatic generation of route sketches. In: *Proceedings of the 18th International Symposium on Graph Drawing (GD'10)*, volume 6502 of *Lecture Notes in Computer Science*, pages 391–392. Springer, 2011. Poster abstract., Joint work with Andreas Gemsa, Martin Nöllenburg, and Ignaz Rutter.

Path schematization for route sketches. In: *Proceedings of the 12th Scandinavian Symposium and Workshop on Algorithm Theory (SWAT'10)*, volume 6139 of *Lecture Notes in Computer Science*, pages 285–296. Springer, June 2010. Joint work with Daniel Delling, Andreas Gemsa, and Martin Nöllenburg.

Parallel computation of best connections in public transportation networks. In: *24th International Parallel and Distributed Processing Symposium (IPDPS'10)*, pages 1–12. IEEE Computer Society, 2010. Joint work with Daniel Delling and Bastian Katz.

Efficient route planning in flight networks. In: *Proceedings of the 9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'09)*, OpenAccess Series in Informatics (OASIcs), 2009. Joint work with Daniel Delling, Dorothea Wagner, and Christos Zaroliagis.

Accelerating multi-modal route planning by access-nodes. In: *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. Joint work with Daniel Delling and Dorothea Wagner.

Engineering time-expanded graphs for faster timetable information. In: *Proceedings of the 8th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'08)*, OpenAccess Series in Informatics (OASIcs), September 2008. Joint work with Daniel Delling and Dorothea Wagner.

Technical Reports

Energy-optimal routes for electric vehicles. Technical Report 2013-06, Faculty of Informatics, Karlsruhe Institute of Technology, 2013. Joint work with Moritz Baum, Julian Dibbelt, and Dorothea Wagner.

Computing and evaluating multimodal journeys. Technical Report 2012-20, Faculty of Informatics, Karlsruhe Institute of Technology, 2012. Joint work with Daniel Delling, Julian Dibbelt, Dorothea Wagner, and Renato F. Werneck.

On d-regular schematization of embedded paths. Technical Report 2010-21, Faculty of Informatics, Karlsruhe Institute of Technology, 2010. Joint work with Andreas Gemsa, Martin Nöllenburg, and Ignaz Rutter.

Path schematization for route sketches. Technical Report 2010-02, Faculty of Informatics, Karlsruhe Institute of Technology, 2010. Joint work with Daniel Delling, Andreas Gemsa, and Martin Nöllenburg.

Parallel computation of best connections in public transportation networks. Technical Report 2009-16, Faculty of Informatics, Karlsruhe Institute of Technology, 2009. Joint work with Daniel Delling and Bastian Katz.

Submitted Journal Articles

User-Constrained Multi-Modal Route Planning. *Journal on Experimental Algorithmics*, April 2012. Joint work with Julian Dibbelt and Dorothea Wagner.

Anhang C

Deutsche Zusammenfassung

Die Routenplanung in Transportnetzwerken ist ein Problem, das aus dem heutigen Alltag nicht mehr wegzudenken ist und eine Vielzahl interessanter Anwendungsfälle umfasst. Die bekannteste Anwendung ist wahrscheinlich das Navigationsgerät in Autos. Weitere Beispiele sind verschiedenen Kartendienste im Internet und Fahrplanauskunftssysteme, wie das der Deutschen Bahn. Die zugrundeliegenden Algorithmen müssen dafür in möglichst kurzer Zeit, zu einer vom Benutzer gestellten Anfrage, optimale Lösungen berechnen.

Ein gängiger Lösungsansatz modelliert das Transportnetzwerk als Graphen, dessen Kantengewichte die zu optimierende Metrik (Reisezeit, Distanz, usw.) repräsentieren. Dijkstras klassischer Algorithmus von 1959 ist dann in der Lage eine beweisbar optimale Route zwischen zwei Knoten in diesem Graphen zu berechnen. Leider ist dieser auf realistischen Eingaben zu langsam, um für interaktive Anwendungen praktikabel zu sein. Daher wurde in den vergangenen Jahren daran geforscht, wie sich Dijkstras Algorithmus (mit Hilfe einer Vorberechnungsphase) beschleunigen lässt. Die schnellsten dabei entstandenen Verfahren sind bis zu sieben Größenordnungen schneller als Dijkstras Algorithmus.

Fast alle dieser sehr effizienten Verfahren sind jedoch für Straßennetzwerke mit Reisezeitmetrik ausgelegt. Zwar ließen sie sich prinzipiell auf andere Netzwerke, wie beispielsweise öffentliche Verkehrsnetze, übertragen, verlieren dort allerdings ihre gute Performanz. Des Weiteren ist man in solchen Netzwerken oftmals an komplexeren Anfragetypen interessiert. So möchte man beispielsweise nicht nur die (einzige) schnellste Verbindung zu einer vom Benutzer festgelegten Abfahrtszeit berechnen, sondern eine *Menge* optimaler Verbindungen über einen ganzen Zeitraum. Betrachtet man mehrere Arten von Transportnetzwerke zusammen (Straße, Fußgänger, öffentlicher Verkehr, usw.), so spricht man von *multimodaler* Routenplanung. Es ist klar, dass die Berechnung optimaler Routen in solchen Netzwerken mindestens die Herausforderungen der jeweiligen Teilnetzwerke beinhaltet. Darüber hinaus hat man

noch das Problem, dass man die einzelnen Modalitäten *sinnvoll* kombinieren muss. Zum Beispiel kann es unerwünscht sein, den Benutzer aufzufordern, zwischen zwei Zugfahrten ein privates Auto zu benutzen.

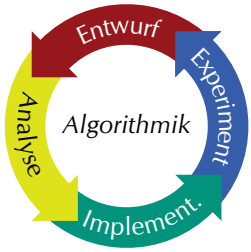


Abbildung C.1. Prinzip *Algorithm Engineering*.

Diese Arbeit setzt sich mit den oben genannten erweiterten Problemstellungen auseinander und führt neue, effiziente algorithmische Verfahren für diese ein. Dabei nutzen die Verfahren explizit die strukturellen Eigenschaften der jeweiligen Verkehrsnetze aus. Das methodische Vorgehen beruht dabei auf dem *Algorithm Engineering* [San09,MS10,SW13] (siehe auch Abbildung C.1), das sich grob als Kreislauf aus Algorithmenentwurf, theoretischer Analyse (bzgl. Korrektheit und Laufzeit), Implementierung und einer experimentellen Auswertung, beschreiben lässt. Grob lässt sich der Inhalt der Arbeit in die folgenden vier Teile gliedern: *Routenplanung für öffentlichen Verkehr*, *multimodale Routenplanung*, *Metrik-unabhängige Routenplanung in Straßennetzwerken* und *Berechnung von Joggingrouten*. Im Folgenden wird auf die jeweiligen Teile genauer eingegangen.

Routenplanung für öffentlichen Verkehr. Bei der Routenplanung auf öffentlichen Verkehrsnetzwerken ist die Eingabe ein *Fahrplan*. Dieser definiert Halte und Fahrten (Züge, Busse, usw.), welche Folgen von Halten zu bestimmten Zeiten abfahren. In dieser Arbeit wird ein neuer, paralleler (Multikern-) Algorithmus eingeführt, der sogenannte *One-To-All-Profilanfragen* berechnet. Bei diesen ist das Ergebnis eine Menge optimaler (bezüglich der Reisezeit) Reisepläne für eine ganze Zeitperiode, und zwar von einem Halt zu *allen* anderen im Netzwerk. Der Algorithmus ist graphbasiert und nutzt geschickt aus, dass sich Routen gegenseitig dominieren, wodurch der Suchraum deutlich reduziert werden kann. Die resultierenden Laufzeiten sind praktikabel, selbst für 24-Stunden-Perioden auf dichten städtischen Nahverkehrsnetzen. Das ermöglicht sogar für eine Teilmenge der Halte eine Distanztabelle vorzuberechnen, welche ausgenutzt werden kann, um den gleichen Algorithmus weiter zu beschleunigen, falls man nur in die optimalen Verbindungen zwischen *Paaren* von Halten interessiert ist.

Neben der Reisezeit ist ein weiteres, mindestens genauso wichtiges Kriterium, die Anzahl der Umstiege. Um dem Benutzer eine sinnvolle Menge an Alternativrouten zu präsentieren, wird in dieser Arbeit das Problem betrachtet, multikriterielle Routen zu berechnen, d. h. *Pareto-Mengen* von nicht-dominierenden (bzgl. Reisezeit und Anzahl Umstiege) Routen. Vor dieser Arbeit war hier der Stand der Technik, erweiterte Versionen von Dijkstras Algorithmus zu benutzen. Diese sind jedoch recht langsam. Die Arbeit führt einen neuen, effizienteren Ansatz ein, der direkt auf dem Fahrplan operiert. Im Gegensatz zu Dijkstras Algorithmus benötigt er weder einen Graphen noch eine Priority-Queue. Stattdessen nutzt er aus, dass Züge auf wohldefinierten Routen fahren, wodurch sich ein dynamisches Programm konstruieren lässt, das sukzessiv die Pareto-Lösung aufbaut. Der Algorithmus ist sehr Cache-effizient und

um eine Größenordnung schneller als bisherige (Graphen-basierte) Verfahren. Anfragen in sehr dichten Netzwerken können so in wenigen Millisekunden beantwortet werden. Der Algorithmus lässt sich außerdem effizient parallelisieren und um weitere Kriterien erweitern. Da er auf keine Vorberechnungen angewiesen ist, lässt er sich unmittelbar für dynamische Szenarien einsetzen, um Verspätungen oder Zugausfälle direkt bei der Routenberechnung zu berücksichtigen.

Multimodale Routenplanung. Ein zweiter Aspekt der Arbeit beschäftigt sich mit der multimodalen Routenplanung, bei der man an *integrierten* Lösungsverfahren, die verschiedene Verkehrsmittel sinnvoll kombinieren, interessiert ist. Ein gängiger Ansatz zulässige Sequenzen von Verkehrsmitteln zu definieren, ist das sogenannte *Label-Constrained Shortest-Path* Problem, bei dem man die Sequenzen über reguläre Sprachen spezifiziert. Eine Variante von Dijkstras Algorithmus auf einem aus den Teilnetzwerken zusammengesetzten Graphen berechnet zwar beweisbar optimale Routen, ist jedoch in der Praxis zu langsam. Die Arbeit präsentiert einen schnellen Ansatz, der auf dem Konzept der Knotenkontraktion basiert und die Eingabe so vorverarbeitet, dass beweisbar die optimalen Lösungen für alle möglichen Verkehrsmittel-Sequenzen erhalten bleiben. Dadurch kann die Sequenz vom Benutzer bei der Anfrage spezifiziert werden. Bisherige Verfahren haben, im Gegensatz dazu, die Sequenz bereits in der (aufwendigen) Vorberechnungsphase fixiert.

Manchmal kann (oder will) der Benutzer jedoch keine Aussage über die erlaubten Verkehrsmittel-Sequenzen treffen. Stattdessen wäre es wünschenswert, eine Menge von *sinnvollen* Alternativrouten (mit unterschiedlichen Verkehrsmittel-Sequenzen) zu berechnen, aus denen der Benutzer wählen kann. Aus diesem Grund wird in der Arbeit die Kombination von multimodaler mit multikriterieller Routenplanung betrachtet. Statt Verkehrsmittel-Sequenzen zu beachten, wird für jede Verkehrsart ein *Bequemlichkeitskriterium* identifiziert. Mit Hilfe dieser Kriterien werden dann Pareto-Mengen von Alternativrouten berechnet. Eine große Herausforderung hierbei ist, dass diese Mengen hunderte, insignifikante Lösungen enthalten können. Diese erhöhen (unnötigerweise) die Berechnungsdauer und sind für den Benutzer kaum von Bedeutung. Daher wird basierend auf unscharfer Mengenlehre (Fuzzy Set Theory) ein *unscharfes Dominanzkriterium* benutzt, das zuverlässig die k signifikantesten Routen aus der Pareto-Menge extrahieren kann. Diese könnten dann dem Benutzer gezeigt werden. Außerdem werden zur Beschleunigung der Suche heuristische Algorithmen vorgestellt, die die Pareto-Mengen bereits im Laufe der Berechnung klein halten, indem als insignifikant klassifizierte Lösungen entfernt werden. Die Qualität der Heuristiken wird mit einem konsistenten Qualitätsmaß evaluiert.

Metrikunabhängige Routenplanung in Straßennetzwerken. In diesem Teil betrachten wir nochmals das (klassische) Problem, optimale Routen in *Straßennetzwerken* zu berechnen. Der Fokus liegt dabei allerdings auf der Vorberechnungsphase. Bei den

existierenden effizienten Algorithmen führt eine Änderung der Metrik im Graphen zu einer Invalidierung der vorberechneten Daten, wodurch diese neu berechnet werden müssen, was aufwendig sein kann. Der vorgestellte Algorithmus basiert auf dem (bekannten) Konzept von Multilevel-Overlaygraphen. Die grundlegende Idee ist dabei, die Vorberechnung in zwei Stufen zu unterteilen. In einer Metrik-unabhängigen Stufe wird mit Hilfe eines Graph-Partitionierers die Topologie der Overlaygraphen festgelegt. In der Metrik-abghängigen Stufe werden basierend auf der Topologie, die Kantengewichte der Overlaygraphen ausgerechnet. Damit kann eine neue Metrik in wenigen Sekunden integriert werden. Dies ermöglicht neue Anwendungen, wie das Einbinden von Echtzeit-Staudaten oder die Unterstützung personalisierter Metriken.

Berechnung von Joggingrouten. Die letzte Problemstellung betrachtet das Berechnen „guter“ *Joggingrouten*. Hier ist die Eingabe ein Startpunkt sowie die gewünschte Länge der Route. Ziel ist es, einen Kreis in einem Fußgängernetzwerk zu berechnen, der die gewünschte Länge annähert und den Startknoten enthält. Zudem müssen sogenannte weiche Kriterien berücksichtigt werden. Beispielsweise soll die Tour eine einfache Form haben, möglichst schöne Gebiete (z. B. Parks und Wälder) durchqueren und wenig Strecke doppelt ablaufen. Für das Problem werden zwei Lösungsansätze vorgestellt. Der Erste beruht auf der Intuition sukzessiv eine gegebenen Tour um eine der angrenzenden Facetten im Graphen zu erweitern. Der zweite Ansatz überträgt die geometrische Intuition bei der Konstruktion von gleichseitigen regelmäßigen Polygonen auf Graphen, um so Touren zu erhalten. Der Algorithmus lässt sich parallelisieren und ist inhärent in der Lage mehrere verschiedene Alternativtouren auf einmal zu berechnen.