Engineering Route Planning Algorithms for Battery Electric Vehicles

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

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

genehmigte

Dissertation

von

Moritz Baum

aus Eutin

Tag der mündlichen Prüfung:	4. Juli 2017
Erste Gutachterin:	Prof. Dr. Dorothea Wagner
Zweiter Gutachter:	Prof. Dr. Stefan Funke

Acknowledgements

This work would not have been possible without the help and support of others. First and foremost, I am deeply grateful to my advisor Dorothea Wagner for granting me a position in her group, where I could benefit from her guidance and work in a pleasant atmosphere. I thank the Federal Ministry of Economics and Technology and the European Commission for partially funding my research under grant agreement numbers 01ME12013 (project iZeus), 288094 (project eCOMPASS), 609026 (project MOVESMART), and 621133 (project HoPE). I also thank the respective consortia for contributing to the success of each project. I am very grateful to Jürgen Branke for giving me the opportunity to take my first steps in academia with his group at the University of Warwick, England. Further, I would like to thank Stefan Funke for acting as second referee of this thesis.

I am very much obliged to all coauthors of our joint publications during the past few years, Reinhard Bauer, Thomas Bläsius, Christian Bräuer, Valentin Buchhold, Julian Dibbelt, Andreas Gemsa, Lorenz Hübschle-Schneider, Alexander Kleff, Thomas Pajor, Ignaz Rutter, Jonas Sauer, Frank Schulz, Dorothea Wagner, Franziska Wegner, and Tobias Zündorf, for plenty of fruitful discussions and inspiring collaborations. I owe gratitude to all my current and former coworkers for being part of an enjoyable and productive working environment. I would like to highlight my office mates Julian Dibbelt and Sascha Gritzbach, who offered help or distraction whenever needed. As one among many great memories, I will never forget the unique experience of our night shift ending in a paper submission at 5:45 am (15 minutes before the strict deadline). No less am I thankful to all current and former members of our administrative staff, Lilian Beckert, Bernd Giesinger, Ralf Kölmel, Laurette Lauffer, Simone Meinhart, and Elke Sauer, for all their help in our daily struggles at work. I especially thank all current, former, and honorable members of the Escorial committee, Obstfreunde, our "saboteurs", as well as anyone joining our regular football games. I will also cheerily remember the board games accompanied by lots of diced cheese, movies playing at double speed in "Cinema 301", and countless other spare time activities.

Next, I thank Lukas Barth, Valentin Buchhold, Benjamin Niedermann, Franziska Wegner, and Tobias Zündorf for proof reading parts of this thesis and for many helpful comments and suggestions. I also owe gratitude to Thomas Bläsius, who wrote the template I used for this thesis. Moreover, I want to express my deep appreciation to everyone not mentioned explicitly by name who supported or contributed to this work in any way. Last but certainly not least, I would like to thank all my friends, my family, and Anna for their constant love and support throughout the past years.

Abstract

In this thesis, we study route planning algorithms designed to fulfill the specific requirements of battery electric vehicles. First and foremost, battery electric vehicles typically employ a rather limited cruising range. Given that charging the battery is time consuming and charging stations are scarce, careful route planning that prevents battery depletion during a ride is of utmost importance. Therefore, we propose algorithms for quickly finding routes that help the user in reliably reaching a destination before the battery fully depletes. Moreover, we introduce algorithmic tools for fast and accurate visualization of the remaining cruising range of a vehicle.

Following the paradigm of *Algorithm Engineering*, our studies include algorithm design based on realistic models, thorough analysis of their complexity, and comprehensive experimental evaluation of all proposed techniques. As a result, our approaches achieve good performance in practice, as we demonstrate on a large real-world instance representing the road network of Western Europe.

The thesis is divided into three major parts. In the first part, we aim at routes that minimize energy consumption in order to maximize the vehicle's cruising range. Algorithmic challenges entail handling negative edge costs (to model the ability of *recuperating* energy while driving) and constraints imposed by the vehicle's battery capacity. We show that all problems considered in this part allow for efficient solutions, not only in theory, but also in practice when executed on large-scale instances.

The second part deals with *Constrained Shortest Path* problems, where the objective is to find the fastest route on which a desired target can be reached. We consider two generalizations of this basic problem, by integrating *charging stops* to charge a battery en route and *adaptive speeds* that allow for additional energy savings by passing speed recommendations to the driver (or to an adaptive cruise control unit). We develop algorithms to solve these challenging, $\mathcal{NP}$-hard problems optimally and carefully engineer them to enable fast solutions in practice.

Finally, we develop efficient techniques for accurate visualization of the remaining range of a battery electric vehicle on a road map, which helps a user in identifying reachable targets. This results in two major subproblems: the computation of the reachable subnetwork and the efficient representation of the reachable area for the actual visualization. For each of these subproblems, we propose a plethora of new techniques that outperform previous approaches in terms of both result quality and running time.

Contents

1 Introduction

Electromobility is a cornerstone of sustainable transportation. During recent years, battery electric vehicles (EVs) have become widely available, promising zero local emissions and independence of fossil fuels. Their highly efficient powertrains help keeping energy consumption low, especially in city traffic. Nevertheless, EVs are still a rare sight on streets today. This can be partially explained by the limited battery capacity of most vehicles, paired with the facts that charging stations are much rarer than gas stations and recharging is time consuming. Consequently, drivers tend to focus solely on preventing battery depletion during a ride. This fear of getting stranded, often referred to as *range anxiety* in the literature [Fra+12, Fra+16, THS09], remains a major restraining factor in the consolidation of EVs.

On the other hand, the past decade has seen a great amount of research in the area of route planning in transportation. A plethora of novel approaches emerged, enabling the computation of provably shortest routes in large-scale road networks within milliseconds and below [Bas+16]. As a result, interactive web-based route planning and onboard navigation have become a commodity for millions of users. However, most services focus on the requirements of conventional vehicles using internal combustion engines. When designing such route planning applications for EVs, careful guidance of the user is crucial to overcome range anxiety. Therefore, energy consumption has to be incorporated accurately in routes that are proposed to users. Given that charging stations are scarce, time-consuming charging stops need to be planned in advance. Besides a limited cruising range, there are other substantial differences to vehicles run by combustion engines, such as the ability to recuperate energy when braking. These aspects have to be reflected in any kind of route planning application for EVs. Aside from the demand for realistic models of energy consumption, driving behavior, and environment, this raises numerous algorithmic challenges. In addition to travel time, route planning algorithms for EVs must explicitly take energy consumption into account to provide adequate solutions. Users may then ask for energy-efficient routes, routes with constraints on both travel time and energy consumption, or a visualization of the remaining cruising range.

In this work, we study route planning algorithms that integrate the above modeling considerations to capture the characteristics and needs of EVs. Thereby, route planning applications could assist drivers in many ways to prevent range anxiety and increase driving comfort and efficiency. Computing routes with low energy consumption, possibly subject to time constraints by the user, could help in maximizing the range of an EV. To save energy, onboard navigation might pass instructions regarding speed

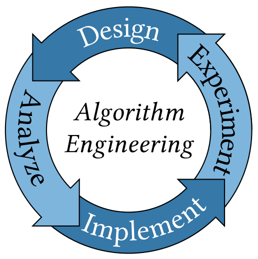


Figure 1.1: The concept of Algorithm Engineering, following Sanders [San09] and Müller-Hannemann and Schirra [MS10].

and driving style to the driver or, going even further, directly to adaptive speed control of a vehicle. If recharging en route is unavoidable to reach the desired destination, route planners could propose charging stops that minimize the overhead in travel time, taking possible detours to reach the charging station and charging time into account. Furthermore, they should instruct the user on how much energy needs to be charged in order safely reach the destination without wasting time. Finally, accurate estimation of the remaining cruising range together with an appropriate range visualization would assist the user in identifying reachable destinations, thereby further reducing range anxiety. Ideally, the navigation system of an EV supports all these features. It turns out that traditional route planning techniques do not cover any of them in an adequate manner, as they focus solely on minimizing travel time of a journey. Therefore, novel approaches are required to address the above requirements properly. This involves the consideration of realistic models of energy consumption and charging infrastructure. At the same time, algorithmic solutions must be fast enough in practice to enable interactive applications.

This thesis introduces algorithmic ingredients of the ideal EV route planning system described above. In particular, we develop algorithms that compute energy-optimal routes, time-constrained shortest paths, routes via charging stops, and accurate visualization of the remaining vehicle range. In doing so, we aim at solutions with good performance, not only in theory, but also in practice when executed on large, realistic inputs. We follow the principle of Algorithm Engineering [MS10, San09, SW11], where algorithm design is followed by theoretical analysis and implementation. Insights from experimental evaluation on real-word instances may then trigger a new cycle of refining design, analysis, implementation, and experiments; see Figure 1.1 for an illustration of this well-established paradigm. As a result, algorithmic approaches presented in this work are substantiated by theoretical guarantees on result quality and worst-case running time, but also by experiments in challenging, realistic settings. The thesis is arranged into three major parts, dealing with energy-optimal routes, time-constrained path computation, and range visualization, respectively. Each algorithmic part is followed by an extensive experimental study, to demonstrate the practicability of our approaches. In the following Section 1.1, we highlight the key contributions of this thesis in more detail. Afterwards, Section 1.2 outlines the remainder of this work.

1.1 Main Contributions

The major contributions of this thesis are separated into three parts. First, new approaches for energy-optimal routes are presented in detail in Chapter 4. Second, Constrained Shortest Path algorithms that broaden the state-of-the-art are discussed in Chapter 5. Third, we propose novel algorithms for efficient range visualization in Chapter 6. We briefly summarize the main results of each of these parts in turn.

Energy-Optimal Routes for Battery Electric Vehicles. As our first contribution, we deal with the important algorithmic problem of computing routes that minimize energy consumption of an EV. Any such approach must cope with specific properties. For example, *recuperation* (i. e., regenerative braking) enables conversion of kinetic energy, so that it can be stored in the battery while driving. This corresponds to *negative costs* in terms of energy consumption. Moreover, *battery capacity constraints* impose limits on the state of charge (SoC) of an EV, restricting its range and the amount of energy that can be recuperated. These restrictions can be captured by *profiles* modeling the interdependence of battery constraints and energy consumption as a special form of piecewise linear functions [EFS11]. These functions are relevant in various query scenarios and a crucial ingredient of speedup techniques for energy-optimal routing. On the theoretical side, we prove that such profiles have linear complexity, much in contrast to conceptually similar profiles in time-dependent routing [FHS14]. We examine different methods to handle negative edge costs, such as potential shifting [Joh77], which enable variants of Dijkstra's algorithm and a stopping criterion for searches. We also show how these methods facilitate efficient *profile search.*

Furthermore, we consider energy-optimal routes with intermediate stops at charging stations, to recharge the battery. Unlike previous studies [GP14, SBW12, Sto12a], we do not assume that using a charging station always results in a fully recharged battery. Instead, we allow the charging process to be *interrupted* beforehand to save energy. We show that the problem can be solved by a label-setting algorithm resembling bicriteria search [Han80]. Building upon our theoretical findings, we also derive a conceptually simple polynomial-time algorithm. We propose a heuristic variant that is easy to implement, and carefully integrate it with the Contraction Hierarchies (CH) algorithm [EFS11, Gei+12b] and A* search [HNR68].

Moreover, exploiting multilevel overlay graphs [JP02, SWZ02], we extend the Customizable Route Planning (CRP) approach of Delling et al. [Del+17] to our scenario in a sound manner. This includes the integration of profile search into preprocessing and the nontrivial adaptation of bidirectional search to respect battery constraints. Thereby, we achieve fast (metric-dependent) preprocessing of the *whole* network, allowing flexible updates due to, e. g., hourly weather forecasts or refinements of the underlying consumption model (as is necessary when machine learning approaches are used to improve the model [GM17, Mas+14]). We propose several query algorithms, from which the most sophisticated variant simultaneously, and in parallel, employs a backward search that helps bound the forward search in order to "guide" it toward the target, while respecting battery constraints.

In a thorough experimental study, we demonstrate that our customizable method exhibits faster query times than known approaches, while improving metric-dependent preprocessing time by three orders of magnitude. Regarding the more complex setting involving charging stops, our practical algorithm formally drops correctness, but *always* finds the optimal solution in our tests. It computes even long-distance routes with charging stops in less than 300 ms. We also show that our speedup techniques scale excellently with the available cruising range. This makes our algorithms robust to future developments in increasing battery capacities.

Constrained Shortest Path Problems. Our second major contribution considers different variants of Constrained Shortest Path (CSP) problems [HZ80] for EVs. Since battery capacities are limited, fastest routes are often *infeasible*. Instead, users are interested in fast routes where the energy consumption does not exceed the battery capacity. For that, users may drive below the posted speed limits to find attractive solutions that save energy, but still use major roads (such as motorways). Hence, route planning should provide both path *and* speed recommendations. Also, stops at charging stations may be inevitable. Careful route planning that incorporates charging stops is crucial in this case, as charging stations are scarce and recharging is time consuming. We address two problem settings that are relevant in this scenario.

First, we introduce an $\mathcal{NP}$-hard variant of the CSP problem that includes *realistic* models of charging stops for EVs. We are interested in routes that, while respecting battery constraints, minimize *overall* trip time, including time spent at charging stations. Existing approaches presume that recharging takes constant time and always results in a fully charged battery to simplify the problem [GP14, SBW12, Sto12a]. By contrast, our solution is flexible enough to handle different types of stations accurately, such as battery swapping stations, superchargers, and regular stations with various charging powers. In particular, we allow *partial* recharging to save time.

Second, we take *adaptive speeds* into consideration, allowing drivers to deliberately reduce speed to save energy. Hence, route guidance should incorporate speed recommendations. To tackle the resulting $\mathcal{NP}$-hard optimization problem [HZ80], previous works trade accuracy of the underlying model or correctness for practical running times [Fon13, GP14, HF14]. Instead, we present a novel algorithmic framework to compute *optimal* constrained shortest paths that uses realistic physical models, takes adaptive speeds into account, and respects battery constraints, i. e., ensures that the SoC at the source suffices to reach the target.

Both settings described above, namely, shortest feasible paths including charging stops or adaptive speeds, result in challenging $\mathcal{NP}$-hard problems. As a first step, we solve them with (exponential-time) extensions of the well-known bicriteria shortest path algorithm [Han80]. In the first problem setting involving charging stops, the efficient modeling of infinitely many combinations of charging durations at different stations poses a notable challenge. We resolve it by identifying, during the search, a finite number of choices that provably include the optimal one. When integrating adaptive speeds, we obtain, for each road segment, a *function* that maps travel time to energy consumption based on a realistic consumption model. We derive operations to efficiently *link* such functions in order to get a functional representation of the energy consumption subject to time spent on a route—a crucial ingredient to our basic algorithm for solving the problem.

To improve running times of the basic approaches, we propose speedup techniques based on A* search [HNR68] and CH [Gei+12b], which are naturally combined for further speedup. A particularly challenging aspect is the computation of shortcuts in the presence of adaptive speeds, as they must store bivariate functions to capture the constraints of our model. For faster queries, we propose heuristic approaches that offer high (empirical) quality.

A comprehensive experimental evaluation on detailed and realistic data shows that, even though we are dealing with $\mathcal{NP}$-hard problems, careful engineering and the incorporation of speedup techniques allow us to optimally solve both problems in about a minute on average, even for long-distance queries on continental road networks. On sensible instances, queries with realistic journey duration are answered even faster, within seconds and below. Our heuristic methods provide high-quality solutions, while query times drop to tens of milliseconds, enabling even interactive applications. Thereby, our approaches clearly outperform and broaden the state-of-the-art.

Range Visualization. The third chief contribution concerns algorithms to visualize the remaining cruising range of an EV. In a more general setting, *isocontours* in road networks represent the area that is reachable from a source within a given resource limit. Besides range visualization for EVs, there is a wide range of other practical applications for isocontour visualization, such as urban planning [GBI12] and geomarketing [Efe+13b]. We study the problem of computing accurate isocontours in realistic, large-scale networks. Efficient performance of our techniques is both proven in theory and demonstrated in practice on large, realistic instances.

As a first subproblem, we deal with the identification of the region in range from a given source within a certain resource limit (e. g., the current SoC). There has been little research on fast algorithms for large-scale inputs. Also, existing approaches [EP14, GBI12] tend to compute more information than necessary in our scenario. Using a more compact representation of the reachable subgraph, we propose several techniques that enable fast computation of its boundary and are easy to parallelize. We describe a new algorithm based on CRP [Del+17, DW15] and a faster variant of the best available technique isoGRASP [EP14], exploiting that not all information computed by the original method is required for visualization purposes. Then, we introduce novel approaches that combine graph partitions with variants of PHAST [Del+13b], a technique originally designed for fast computation of batched shortest paths. We also discuss parallelization for further speedup.

Regarding actual visualization, we propose isocontours represented by *polygons* that separate reachable and unreachable components of the network. To allow for fast rendering in practice, we aim at *minimizing* the number of segments of these polygons. Since the resulting problem is not known to be solvable in polynomial time, we introduce several heuristics that run in (almost) linear time and are simple enough to be implemented in practice. All approaches compute isocontours that are *exact* in the sense that they contain exactly the subgraph reachable within the given resource limit, while aiming at a small number of segments. A key ingredient is a new practical linear-time algorithm for minimum-link paths in simple polygons.

Experiments in a challenging realistic setting demonstrate excellent performance of our algorithms in practice. Our portfolio of techniques for computing the reachable subgraph offers different tradeoffs in terms of customization time, space requirements, and query performance. All of them compute the reachable subgraph in less than 100 ms, significantly faster than previous methods. Parallelization reduces running times further, to only a few milliseconds. As for isocontour visualization, our heuristics compute near-optimal solutions within tens of milliseconds, even for long ranges. In total, we achieve query times that enable interactive range visualization for online map services.

1.2 Thesis Outline

We briefly outline the structure of the remainder of this thesis. We point out that parts of this work appeared in previously published scientific journals, proceedings, and reports [Bau+13a, Bau+13b, Bau+15a, Bau+15b, Bau+16a, Bau+16b, Bau+16c, Bau+16d, Bau+17a, Bau+17b, Bau+18]. We also indicate these publications below.

Chapter 2 provides a thorough overview of related work. We distinguish practical approaches for classic problem settings in route planning, discussed in Section 2.1, and more recent works discussing models and algorithms geared towards the requirements of EVs, covered in Section 2.2.

Chapter 3 formalizes basic concepts and recaps known algorithms this thesis builds upon. We introduce necessary notation from graph theory (Section 3.1) and computational geometry (Section 3.2). In Section 3.3, we describe variants of the shortest path problem, Dijkstra's algorithm, and relevant speedup techniques. Section 3.4 presents the experimental setup used in our evaluations.

Chapter 4 considers energy-optimal routes for EVs. In Section 4.1, we formalize the notion of battery constraints, SoC profiles, and energy-optimal routes. Further, we investigate the complexity of these profiles. Section 4.2 discusses different basic approaches to compute energy-optimal paths or profiles. Section 4.3 extends the problem setting to routes with charging stops. We examine the complexity of the resulting problem and provide algorithmic solutions. Section 4.4 presents a speedup technique that quickly computes energy-optimal routes, but at the same time allows frequent changes in the cost function. Finally, Section 4.5 contains our experimental study, while Section 4.6 concludes the chapter. This chapter is based on joint work with Julian Dibbelt, Thomas Pajor, Jonas Sauer, Dorothea Wagner, and Tobias Zündorf [Bau+13a, Bau+13b, Bau+17b].

Chapter 5 investigates CSP problems for EVs. We formally introduce the basic problem setting in Section 5.1. Section 5.2 extends this problem to routes via charging stations. We present algorithms and speedup techniques to solve it optimally, as well as heuristics. Section 5.3 considers another generalization of the basic problem, by taking adaptive speeds into consideration. Again, we discuss techniques to solve the resulting setting, either optimally or by heuristic means. Our experimental study follows in Section 5.4. Section 5.5 closes with final remarks. This chapter is based on joint work with Julian Dibbelt, Andreas Gemsa, Dorothea Wagner, and Tobias Zündorf [Bau+15b, Bau+17a]. It uses insights from joint work with Julian Dibbelt, Andreas Gemsa, Lorenz Hübschle-Schneider, Thomas Pajor, and Dorothea Wagner [Bau+14, Bau+16e].

Chapter 6 deals with the fast computation of isocontours to visualize the range of an EV. The problem and our general approach are described in Section 6.1. Following sections tackle different steps of this approach, namely, computing the subgraph that is reachable from a given position (Section 6.2), the transformation of this subgraph to a geometric representation of its boundary (Section 6.3), and algorithms to compute range polygons for an important special case (Section 6.4) and in the general case (Section 6.5). We present experimental results in Section 6.6 and draw a conclusion in Section 6.7. This chapter is based on joint work with Thomas Bläsius, Valentin Buchhold, Julian Dibbelt, Andreas Gemsa, Ignaz Rutter, Dorothea Wagner, and Franziska Wegner [Bau+15a, Bau+16a, Bau+16b, Bau+16c, Bau+16d, Bau+18].

Chapter 7 concludes the thesis with a brief summary of the key results and insights in Section 7.1. Finally, we provide an outlook on interesting and promising directions of future work in Section 7.2.

2 Literature Overview

This chapter gives a comprehensive overview of existing work related to this thesis. We review existing algorithmic approaches towards route planning in general and for EVs in particular. We begin with known techniques designed for classic shortest path problems in road networks (with vehicles running on combustion engines in mind) in Section 2.1. We focus on practical approaches that achieve good (empirical) query performance by means of preprocessing. For a more complete literature overview, we refer the reader to recent survey articles [Bas+16, FSR06, Som14]. Afterwards, Section 2.2 summarizes algorithmic publications in the context of route planning for EVs. We discuss literature related to the major aspects of this thesis, namely, the computation of energy-optimal routes, constrained shortest paths, routes with charging stops, and range visualization.

While the focus of this chapter is on route planning algorithms for (single) EVs, we remark that other lines of research deal with aspects of electromobility that have no immediate ties to our work, such as reasonable placement of new charging stations [APV16, FNS15, FNS16, Hes+12], the integration of EVs into car sharing applications [Bra+16], as well as management and routing of fleets of EVs [PJL16, YHZ16].

2.1 Speedup Techniques

Route planning in road networks in general has seen substantial algorithmic progress over the past years. Classic route planning approaches make use of a graph-based representation of the underlying transportation network, where scalar edge weights correspond to, e. g., travel times. A shortest path is then found by Dijkstra's well-known algorithm [Dij59]. However, despite its low asymptotic complexity, Dijkstra's algorithm is too slow in practice when run on large, realistic input. Therefore, a great amount of research has focused on improving its practical performance. In particular, *speedup techniques* [Bas+16] accelerate online shortest-path queries with data *preprocessed* in an offline phase, with different benefits in terms of preprocessing time and space, query time, and simplicity. Most were initially developed for static edge costs representing travel times in a road network and later extended to more complex scenarios, such as time-dependent edge costs.

Bidirectional search [Dan63, Dre69, Nic66] improves the running time of Dijkstra's algorithm by simultaneously searching from both the source and the target vertex. Another important concept is *goal-directed* search, which attempts to "guide" the search towards the target. In A* search [HNR68], this is achieved by deriving estimates on the distance to the target based on, e. g., Euclidean distance [Poh71, SV86]. A successful variant, ALT (A*, Landmarks, Triangle Inequality) [EP13, GH05, GW05], obtains good potentials from precomputed distances to selected landmark vertices. In Arc Flags [Hil+09, Lau04, Moh+06], on the other hand, the graph is partitioned into multiple regions. Precomputed flags at edges indicate whether they are contained in at least one shortest path to a specific region. In a query, this enables Dijkstra's algorithm to prune the search at edges without a set flag for the target cell.

Other techniques use precomputed distances between important vertices for faster queries. Many employ *overlay edges* or *shortcuts* that maintain shortest path distances, allowing the search to skip parts of the graph. In Reach [GKW09, Gut04, MCB14], each vertex stores the maximum length of (the shorter end of) a shortest path passing through that vertex, called its *reach value*. Based on this information, the search can be pruned at vertices with low reach value. Transit Node Routing (TNR) [ALS13, Bas+07, BFM09] uses a distance table between a subset of important vertices for speedup. In Hub-Based Labeling (HL) [Abr+11, Abr+12b, AIY13, Coh+03, Del+14a, Del+14b, Gav+04], each vertex stores a set of *hub vertices*, together with corresponding distances. The distance between arbitrary pairs of vertices is obtained from the intersection of their hubs. Contraction Hierarchies (CH) [DSW16, Gei+12b] iteratively *contract* vertices in increasing order of (heuristic) importance during preprocessing, maintaining distances between all remaining vertices by adding *shortcut* edges where necessary. The CH query is then bidirectional, starting from source and target, and proceeds only from less important to more important vertices on the original graph augmented with shortcuts. Multilevel Dijkstra (MLD) [Del+09, HSW09, JP02, SWW00, SWZ02] adds shortcuts between separators of a multilevel partition of the input graph. Queries run on an *overlay graph* utilizing these shortcuts and the subgraphs of the original graph induced by cells containing the source and the target.

Combining CH and ALT, the technique Core-ALT [Bau+10] contracts all but the most important vertices (e. g., the top 1 %), performing ALT on the remaining *core* graph. This approach can also be adapted to more complex scenarios, such as edge constraints that model, e. g., the maximum allowed height or weight of a vehicle [Gei+12a]. Combinations of other aforementioned techniques are possible as well, typically by integrating a goal-directed algorithm (ALT or Arc Flags) with an approach that employs precomputed distances, such as Reach [Bau+10, GKW09], TNR [Bau+10], CH [Bau+10, BD09], or MLD [EPV15, Hol+06].

Dynamic and Time-Dependent Route Planning. To enable relatively fast integration of (local) traffic updates, *dynamic* variants of CH [Gei+12b], ALT [DW07], and Arc Flags [DAn+12] have been studied. Going even further, more recent techniques based on CH [DSW16], ALT [EP13], and MLD [Del+17] were developed that allow an additional *customization* phase after preprocessing, providing fast updates of the *whole* network to quickly incorporate global traffic information or user preferences. In particular, a recent variant of MLD called Customizable Route Planning (CRP) [Del+17] is explicitly designed to work with arbitrary metrics and is capable of integrating new cost functions within seconds and below. Applying the same concept to CH, Customizable Contraction Hierarchies (CCH) are introduced by Dibbelt et al. [DSW16]. As an important ingredient of their preprocessing routine, all customizable approaches rely on graph partitioning algorithms tailored to road networks, which compute partitions with balanced cell sizes and small separators [Del+11b, HS16, SS12, SS15].

While dynamic techniques aim at fast integration of unforeseen changes in traffic or user-dependent modifications, *time-dependent* route planning takes historic knowledge about traffic patterns into account. Scalar edge costs are replaced by (periodic) functions, mapping departure time to travel time along an edge. Dijkstra's algorithm can be extended to *earliest arrival (EA)* queries in this scenario, computing the shortest path subject to departure time at the source [CH66, Dre69]. A *profile query* asks for a functional representation of travel time between two vertices for *any* departure time. Such functions may have superpolynomial complexity [FHS14], but can be computed by an (output-sensitive) variant of Dijkstra's algorithm [Dea04, DW09]. Both EA and profile queries can also be handled by (approximate) landmark-based distance oracles [Kon+15, Kon+16, KZ16] and by extensions of speedup techniques mentioned above, including ALT [DW07, Nan+12], CH [Bat+13], as well as (core-based) combinations of different techniques [Del11, DN12]. Batz et al. [BS12] consider a variant of time-dependent shortest paths, where costs are the sum of time-dependent travel times and independent scalar costs (e. g., energy consumption). Even though this makes the problem $\mathcal{NP}$-hard, a heuristic variant of time-dependent CH enables query times in the order of milliseconds for beneficial instances. Finally, variants of Core-ALT [DN12] and CRP [Bau+16f] allow dynamic changes in time-dependent graphs. Extensions of the speedup techniques ALT [Kir+11], TNR [DPW09], HL [Del+15], and CH [Del+13a, DPW15b] are also used in multimodal scenarios, which combine road networks with time-dependent schedule-based public transit [Pyr+08].

Extended Scenarios. Although originally designed for point-to-point queries, both CH and CRP can be adapted to other scenarios. Scanning the hierarchy of vertices in the graph (induced by a vertex order or multilevel partition, respectively) in a final top-down sweep enables *one-to-all* queries: PHAST (PHAST Hardware-Accelerated Shortest Path Trees) [Del+13b] applies this technique to CH, GRASP (Graph Separators, Range, Shortest Path) [EP14] to CRP. For *one-to-many* queries, RPHAST (restricted PHAST) [Del+13b] and reGRASP (restricted GRASP) [EP14] restrict the downward search in an initial *target selection* phase. Furthermore, *many-to-many* queries can be handled by bucket-based approaches [Kno+07], which precompute pairs of target and distance at vertices. This generic approach can be implemented by means of different speedup techniques [DGW11]. Other related query types investigated in the literature include point-of-interest (POI) queries, best-via queries (shortest paths that visit certain types of POIs), and k-nearest-neighbor (kNN) queries. Adaptations of the speedup techniques ALT [EEP16, EPV15], HL [EEP15], CH [FWL12, Gei+10, Gei11], and CRP [DW15, EPV15] to these and other related query types exist. Bucket-based approaches can be extended to such scenarios as well [Abr+12a].

The techniques mentioned above optimize a single criterion, typically travel time. However, Dijkstra's algorithm also extends to scenarios with multiple criteria, by utilizing multi-dimensional vertex labels that represent sets of *Pareto-optimal* paths [Han80, Mar84]. Several variants exist, which employ different expansion strategies [CM82, RE09, Skr00]. Pareto optimization is of particular relevance in the context of routing in public transportation networks [Dib+13, DPW15a, MW06]. Although computing all Pareto-optimal paths is theoretically hard [GJ79, Han80], it is often feasible for such transportation networks in practice [MW01]. For general networks, the recent NAMOA* (New Approach to Multi-Objective A*) algorithm is an extension of A* search to the multicriteria case [MP10, SW91]. This approach was also applied to road networks [MM12] and later parallelized [EKS14, SM13]. Closely related CSP formulations [HZ80, Zie01] ask to find a shortest path that does not exceed a certain resource limit. CH was also adapted to CSP problems [Sto12b]. For the case that the metric is a user-specific linear combination of multiple criteria, extensions of CH are available as well [FLS17, FS13, GKS10].

2.2 Route Planning for Battery Electric Vehicles

We discuss known approaches that tackle problems in the context of route planning for EVs. First, we survey publications dealing with algorithms that minimize energy consumption on a trip. Often, these works also examine physical energy consumption models. Second, we cover literature on more complex CSP settings, where both travel time and energy consumption are constrained. Third, existing techniques that take charging stops into account are summarized. Finally, we review related work on estimation and visualization of the remaining cruising range of an EV.

Energy-Efficient Routes. Using energy consumption as routing metric may result in negative cost values for some edges, though physical constraints prohibit negative cycles. A label-correcting variant of Dijkstra's algorithm can be applied to compute the shortest path in such a scenario, however, it can have exponential running time [Joh73]. The well-known algorithm of Bellman-Ford-Moore [Bel58, For56, Moo59] handles negative edge costs and has quadratic time complexity. Algorithms with better worst-case bounds in the presence of negative edge costs exist for certain graph classes [CF14, DI10, KMW10, Yen70]. Moreover, there are approaches for fast detection of negative cycles in graphs [Che+10]. To get rid of negative edge costs, one can use a technique called *potential shifting* [Joh77]. This reenables Dijkstra's (label-setting) algorithm.

Quite a few works explicitly consider EVs and attack the problem of computing routes that minimize energy consumption. In this setting, algorithms must ensure that the battery of an EV does not fully deplete, but energy is recuperated when braking (e. g., going downhill), though only up to the limited capacity. These additional *battery constraints* have to be checked during route computation.

Artmeier et al. [Art+10a, Art+10b] handle negative costs with the Bellman-Ford-Moore algorithm and label-correcting variants of Dijkstra's algorithm. While the former turns out to be too slow in practice, the latter achieves running times in the order of seconds on a graph of subcountry scale in their implementation. Battery constraints are checked explicitly in the algorithm during edge relaxation, without affecting its asymptotic complexity. Using a simple physical energy consumption model, Sachenbacher et al. [Sac+11] combine potential shifting (obtained directly from the consumption model) with goal-directed search to get a factor of 3 speedup over their previous label-correcting approach.

Eisner et al. [EFS11, Sto13] observe that battery constraints can be managed implicitly, by assigning a *consumption profile* to each road segment, which maps current SoC to actual consumption. Thereby, battery constraints are modeled as piecewise linear functions, similar to approaches in time-dependent route planning [Bat+13, Dre69, DW09], but mapping SoC to energy consumption instead. They show that the complexity of the consumption profile of a path is constant. Further, they adapt CH to compute energy-optimal routes in less than 50 ms on large graphs, making their technique the fastest one available for this problem.

Integrating battery constraints into route planning via piecewise linear functions, Schönfelder et al. [SLW14] consider *profile search* to compute optimal consumption for *every* initial SoC between a given pair of vertices, along the lines of time-dependent profiles [Dea04, DW09]. Variants of A* search and CH are proposed, the latter of which has average running times of a few milliseconds on a subcountry-scale road graph. The connection of energy-optimal routing and profile search to the more general concepts of functional and algebraic routing is investigated in a follow-up work [SL15].

Kluge et al. [Klu+13, Klu11] consider energy-optimal routes in a time-dependent scenario, using a detailed physical model and a mesoscopic traffic load model. They propose a search based on Dijkstra's algorithm. Dealing with a rather complex setting, its running time is in the order of minutes, even on small inputs. Heuristic extensions enable faster query times of less than a second.

Models of Energy Consumption. Another line of research covers realistic *consumption models* and their effect on optimal route choices. Typically, these works do not focus on routing algorithms. Masikos et al. [Mas+14] use machine learning to obtain consumption data from real EVs in field tests. Genikomsakis and Mitrentsis [GM17] propose EV consumption models explicitly designed for route planning applications, allowing efficient adaptation of model parameters. Neubauer [Neu10] analyses consumption models and incorporates them into route optimization. Other works examine the effect of model parameters (e. g., road types) and data accuracy (e. g., elevation data samples) on energy consumption models [Asa+16, GAP15, Yao+13, ZY15].

Constrained Shortest Path Problems. Approaches discussed above optimize energy consumption as single criterion. However, energy-optimal routes often exhibit disproportionate detours, as using minor, slow roads saves energy due to less air drag. To reflect this *tradeoff* between travel time and energy consumption, other approaches deal with variants of the $\mathcal{NP}$-hard CSP problem, computing routes that minimize energy consumption of an EV without exceeding some time limit [Jur+14, LSS17, Sto12a], or fastest routes that do not violate battery constraints [WJM13]. As a general observation, problem complexity increases significantly in such multicriteria scenarios. Many works drop correctness to achieve practical performance. Even in traditional route planning (not explicitly designed for EVs), practical *exact* speedup techniques are only known for basic problem variants [FS13, GKS10, Sto12b].

Several works extend bicriteria search [Han80, Mar84] to tackle the CSP problem variants for EVs mentioned above. Liu et al. [LSS17] examine time-dependent energy consumption and propose a label-setting variant of bicriteria search that handles the constraints of their model. They apply a known approximation algorithm [PS08] and a greedy heuristic to answer queries within milliseconds on a country-scale graph. Wang et al. [WJM13] take dynamic traffic changes into account, proposing vehicle-to-vehicle communication for context awareness during route computation. Energy-optimal paths are computed with a variant of A* search. Similarly, Jurik et al. [Jur+14] propose a simple bicriteria extension of A* search. However, neither of the two aforementioned works evaluate running times in detail, as they focus mostly on system architecture. Finally, Storandt [Sto12a] employs CH in a CSP scenario for EVs, which yields provably optimal results and query times in the order of milliseconds on subcountry-scale networks.

Works mentioned so far assume that the speed on a road segment is *fixed* in the sense that it cannot be adjusted by the driver, neglecting the possibility to save a significant amount of energy by reducing the *driving speed* (e. g., on motorways). Flores et al. [Flo+15] consider the problem of planning driving speeds of an EV for a given route in the network, i. e., they only plan speeds, but not the route itself. Lv et al. [Lv+16] introduce a dynamic programming approach to optimally plan the speed of a solar-powered EV. Designed for simulation purposes, it is too slow for interactive applications. Fontana [Fon13] proposes a variant of the CSP problem with the additional requirement of determining velocities for all road segments. Taking additional robustness criteria into account to deal with uncertainty in terms of time and energy consumption, a heuristic approach based on Lagrangian relaxation [AMO93] achieves reasonable running times (few seconds) on small graphs. Hartmann and Funke [HF14] model tradeoffs as continuous *functions*, assuming the driver can go at *any* speed up to a given speed limit per edge. They propose an extension of CH to this scenario, but do not consider battery constraints. Their heuristic query algorithm resorts to sampling the continuous tradeoff functions and requires minutes to answer queries on large networks. Alternatively, *two-phase paths* consist of a fastest and an energy-optimal subpath [GP14], where driving speed along edges is determined by the type of the respective subpath. This allows for reasonably fast queries in the order of seconds (without preprocessing), but results are not optimal in general. Strehler et al. [SMS17] give theoretical insights for a CSP problem including variable speeds for EVs. Most importantly, they develop a fully polynomial-time approximation scheme (FPTAS) [Cor+09] for this problem. Unfortunately, the algorithm is slow in practice. They also propose a heuristic search based on discretized speeds, but do not evaluate their approach.

Other works consider multicriteria optimization in closely related fields. For example, Sun and Zhou [SZ16] consider tradeoffs between monetary costs and travel time in route planning for plug-in hybrid vehicles. De Souza et al. [SRB16] propose a bicriteria shortest path search in traffic assignment. They investigate changes in the traffic flow caused by EV route choices and also take congestion effects into account. Finally, we note that speed planning for vehicles is relevant not only in the context of route planning for EVs, but also in several other areas of transportation, such as vehicle (fleet) routing [QE16], supply chain management [BM16], or aviation [XDP16].

Integrating Charging Stops. Without recharging, large parts of the road network are simply not reachable by an EV, rendering long-distance trips impossible. (For conventional cars, broad availability of gas stations and short refuel duration allow to neglect this in route optimization.) Charging stations have been considered by previous works, often under the simplifying assumptions that the charging process takes *constant* time (independent of the initial SoC) and always results in a *fully* recharged battery [GP14, LLS16, SF12, Sto12a, SZ16]. Then, feasible paths between charging stations are independent of source and target, hence easily precomputed. Routes with a minimum number of intermediate charging stops can then be computed in less than a second on subcountry-scale graphs [SF12, Sto12a]. Smith et al. [SBW12] consider a related problem (motivated by aircraft scheduling rather than route planning for EVs), where certain edges in a network reset resource consumption.

Nevertheless, the simple model used in the above works only applies to battery swapping stations, which are still an unproven technology and business model. For regular charging stations, charging time depends on the desired SoC. Kobayashi et al. [Kob+11] distinguish slow and fast charging stations, but assume that the battery is always fully recharged. They propose a heuristic search based on preselected candidates of charging stations for a given pair of source and target. Their approach takes several seconds on metropolitan-scale routes.

In reality, while nearly linear for low SoC, the charging rate decreases when approaching the battery's limit. Thus, it can be reasonable to only charge up to a *fraction* of the limit. Sweda et al. [SDK17] reflect this behavior by combining a linear with an exponential function for high SoC. Liu et al. [LWL14] model charging between 0 % and 80 % SoC with a linear function, but recharging above 80 % SoC is suppressed altogether. Neither approach was shown to scale to road networks of realistic size. Also, while omitting the possibility of charging beyond 80% might be appropriate for regions well covered with charging stations, it drastically deteriorates reachability in regions with few stations, where recharging to a full battery can be inevitable [Mon+17].

Other works discretize possible charging durations to model different options [HB15, SMS17, WJM13]. This enables search algorithms that closely resemble the well-known bicriteria shortest path algorithm [Han80]. The FPTAS of Strehler et al. [SMS17] mentioned above can also be extended to allow charging stops.

A technique based on dynamic programming to optimize a certain cost function on a route with charging stops is given by Sweda and Klabjan [SK12], extending previous approaches on refueling strategies for conventional cars [KMM11, LGR07]. Similarly, Liang et al. [LLS16] use dynamic programming to solve different routing problems for EVs, including the possibility of battery swaps along the way. Finally, charging stops are considered in several works dealing with vehicle (fleet) routing problems for EVs [Des+16, GS15, Mon+17, PCW16, SSG14, SW18, Yan+15] or scenarios with additional constraints [Adl+16, Ali+14], which are handled by means of mathematical programming or heuristics. Aiming at complex scenarios (often involving optimization for *multiple* vehicles), these approaches do not scale to large input instances.

Range Visualization. Visualization of the remaining cruising range of an EV is another important tool to reduce range anxiety. The area reachable by an EV is represented by its boundary, which is called *isocontour*.

Several works propose systems for accurate range estimation, but typically do not set their focus on fast visualization of isocontours [CBH11, OP14a, OP14b]. A closely related application is the computation of *isochrones*, where the considered resource limit is time instead of energy consumption. There is a wide range of applications for isochrones, including reachability analyses [Bau+08, Gam+11, GBI12, OMS00, SC07], geomarketing [Efe+13b], as well as environmental and social sciences [IBG13].

Some existing algorithms deal with the subproblem of computing the part of the network that is reachable within a given timespan. The MINE (Multimodal Incremental Network Expansion) algorithm [Gam+11] is a search based on Dijkstra's algorithm that computes the reachable part in multimodal networks (including road and public transit). An improved variant, called MINEX (MINE with Vertex Expiration), reduces space requirements [GBI12]. Both approaches work on databases and were later extended to incremental searches, where multiple isocontours with different ranges from a given source are computed [KSG14]. However, due to the lack of preprocessing, running times are prohibitively slow, even on relatively small instances.

Regarding speedup techniques, the boundary of the reachable subnetwork is not known in advance, but part of the *query output*. Hence, target selection (as in one-to-many queries) [Del+13b, EP14] or backward searches [EPV15] are not directly applicable in our scenario. Tesfaye and Augsten [TA16] propose a technique based on graph partitioning for closely related "reachability queries" in public transport networks, but they do not evaluate their algorithm. An extension of the CRP approach [Del+17, DW15] to isochrones is outlined in a patent,[1] however, in a simpler than our intended scenario. Furthermore, the approach was neither implemented nor evaluated. Finally, GRASP can be extended to isochrone queries, but isoGRASP (isochrone GRASP) [EP14] computes distances to all vertices that are reachable from the source, which can be wasteful if only the actual isocontour is required.

Approaches listed above only deal with the computation of the *reachable subgraph* with respect to some resource limit, rather than computing the actual isocontours. Regarding their *visualization*, efficient approaches exist for shape characterization of point sets, such as α-shapes [EKS83], χ-shapes [Duc+08], or Voronoi Filtering [AB99]. However, we are interested in separating subgraphs rather than point sets. A work by O'Sullivan et al. [OMS00] introduces approaches for isocontour visualization based on merging shapes covering the reachable area. Alternatively, one can subdivide the plane into cells and highlight those that intersect the reachable area [KH16]. Marciuska and Gamper [MG10] present two approaches to visualize isochrones in transportation networks. The first transforms a given reachable network into an isochrone by simply drawing a buffer around all edges in range. The second creates a polygon without holes, induced by the edges on the boundary of the embedded reachable subgraph. This algorithm is also used as state-of-the-art in several recent works that consider applications of isocontours [Efe+13a, Efe+13b, GBI12]. However, both aforementioned approaches were implemented on top of databases, providing running times that are too slow for many applications (several seconds for small and medium ranges).

Numerous works present applications that make use of isocontours in the context of urban planning [Bau+08, IBG13, MG10, OMS00], geomarketing with integrated traffic information [Efe+13a, Efe+13b], and range visualization for EVs [CBH11, GBL14, OP14a, OP14b]. For isocontour visualization, these works typically resort to the approach of Marciuska and Gamper [MG10] or less accurate solutions, such as α-shapes or the convex hull of all reachable points.

[1]US Patent Application 13/649,114; `http://www.google.com/patents/US20140107921`

3 Fundamentals

In this chapter, we introduce notation and describe basic concepts used throughout the thesis. In doing so, we assume the reader to be familiar with foundations of computational complexity, such as the concept of $\mathcal{NP}$-hardness and the Bachmann-Landau notation. For backgrounds on these topics, we refer to fundamental literature on theoretical computer science and algorithmics [Cor+09, GJ79, MS08]. Below, we first give basic definitions on graph theory and computational geometry in Section 3.1 and Section 3.2, respectively. In Section 3.3, we formally define variants of the shortest path problem and review Dijkstra's algorithm and possible generalizations. We also recap speedup techniques relevant for our work. In Section 3.4, we describe the experimental setup of our evaluations presented in subsequent chapters.

3.1 Graph Theory

A *(directed) graph* is a tuple $G = (V, E)$ consisting of a finite set V of *vertices* and a set $E \subseteq V \times V$ of ordered pairs of vertices, called *edges*. We denote by $n = |V|$ the number of vertices and by $m = |E|$ the number of edges in the graph. Road networks are typically modeled as directed graphs, where intersections are represented by vertices $v \in V$ and road segments between intersections by edges $e = (u, v) \in E$. In an *undirected graph* $G = (V, E)$, the set $E \subseteq 2^V$ of edges consists of two-element subsets of V, denoted $e = \{u, v\} \in E$, rather than ordered pairs. We also consider *multi-graphs* $G = (V, E)$, where E is a *multiset* of *edges*, i. e., there is a mapping $\mu \colon E \to \mathbb{N}$ denoting the multiplicity of each edge. In other words, multi-graphs may contain parallel (multi-)edges between vertices. If not mentioned otherwise, we assume that graphs are directed and contain no multi-edges in what follows. Definitions given below extend to the undirected case or multi-graphs canonically.

We call u the *tail* and v the *head* of an edge $(u, v) \in E$, and vertices are *neighbors* if they are connected by an edge. Moreover, the edge (u, v) is *incident* to both u and v, it is an *outgoing* edge of u, and an *incoming* edge of v. The vertices u and v are also called *adjacent*. Given a vertex $v \in V$, its *in-degree* is the number of its incoming edges, i. e., the cardinality $|\{u \in V \mid (u, v) \in E\}|$. Similarly, the *out-degree* of v is the number of its outgoing edges. Finally, the *degree* of v is the sum of its indegree and its outdegree.

Cost Functions. A *cost function* $z \colon E \to Y$ on a graph $G = (V, E)$ maps edges $e \in E$ to costs $z(e)$, where the codomain Y typically is a subset of the real numbers, i. e., $Y \subseteq \mathbb{R}$. We simplify notation by defining $z(u, v) := z((u, v))$ for an edge $(u, v) \in E$.

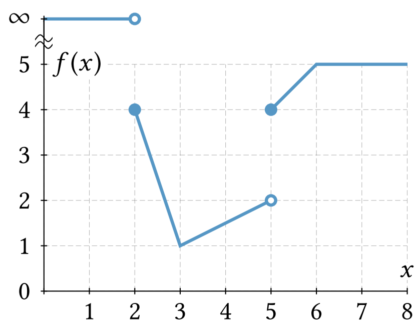


Figure 3.1: The plot of a piecewise linear function f. The function is defined by the sequence $F = [(2,4),(3,1),(5,2),(5,4),(6,5)]$ of breakpoints, with $f(x) = \infty$ for $x < 2$. Note that there is a discontinuity at $f(5) = 4$. Filled circles indicate function values at segment borders.

To give a simple example, we assume that graphs representing road networks come with two cost functions $d\colon E \to \mathbb{R}_{\geq 0}$ and $c\colon E \to \mathbb{R}$ mapping each road segment to its corresponding driving time and energy consumption, respectively. Note that energy consumption may be negative, reflecting the possibility of recuperation.

We also deal with edge costs that are not scalar, to model, e. g., different driving speeds when traversing an edge or energy consumption that depends on the current SoC of an EV. We therefore require *function spaces* $\mathbb{F}$ consisting of functions of the form $f\colon X \to Y$, with domain $X \subseteq \mathbb{R} \cup \{-\infty, \infty\}$ and codomain $Y \subseteq \mathbb{R} \cup \{-\infty, \infty\}$. We often consider *piecewise-defined* functions, i. e., functions defined by multiple *subfunctions*, each applying to certain *subdomains* of X.

A class of piecewise-defined functions that are of particular interest in this thesis are *piecewise linear functions*. A piecewise linear function f is represented by a sequence $F = [(x_1,y_1),\dots,(x_k,y_k)]$ of *breakpoints*, such that $x_i \leq x_j$ for $i < j$. Each breakpoint $(x_i,y_i) \in X \times Y$, with $i \in \{1,\dots,k\}$, is defined by its *x-coordinate* x_i and its *y-coordinate* y_i. Then, f is evaluated by linear interpolation between breakpoints, handling border cases according to the general form

$$f(x) := \begin{cases} \pm\infty & \text{if } x < x_1, \\ y_i + (x - x_i)\frac{y_{i+1}-y_i}{x_{i+1}-x_i} & \text{if } x_i \leq x < x_{i+1} \text{ for some } i \in \{1,\dots,k-1\}, \\ y_k & \text{otherwise;} \end{cases}$$

see Figure 3.1 for an example. Note that we allow the case $x_i = x_{i+1}$ for (at most) two consecutive breakpoints to model discontinuities. Moreover, we abuse notation from set theory for sequences of breakpoints or vertices. For example, an empty sequence is denoted $F = \emptyset$ and represents the function $f \equiv \pm\infty$. (The sign of the infinite value has to be specified in the definition of the function space.)

We often denote by $f_a \equiv a$ the constant function that evaluates to $f_a(x) = a$ for all $x \in X$. A function f is *nonnegative* on some interval $I \subseteq X$ if $f(x) \geq 0$ for all $x \in I$ and *nonpositive* if $f(x) \leq 0$ for all $x \in X$. Similarly, it is *positive* or *negative* if the respective inequalities are strict. We denote by $f^{\min} := \min_{x \in X} f(x)$ the *minimum* value and by $f^{\max} := \max_{x \in X} f(x)$ the *maximum* value of f (provided that these values exist). A function is said to be *invertible* on some interval $I \subseteq Y$ if for all $y \in I$, there is a unique value $x \in X$ such that $f(x) = y$. In this case, the function $f^{-1} \colon I \to X$ with $f^{-1}(f(x)) = x$ is called the *inverse* of f (on I). Another special function is the *identity function*, denoted $\mathrm{id} \colon X \to X$, which evaluates to $\mathrm{id}(x) = x$ for all $x \in X$.

Given two functions f and g defined on the same domain X, we denote by $f + g$ the function obtained by pointwise addition, i. e., $(f + g)(x) := f(x) + g(x)$. The definition of $f - g$ is analogous. The pointwise minimum $\min(f,g)$ of f and g yields their *lower envelope*, which evaluates to $\min\{f(x), g(x)\}$ for all $x \in X$. The *upper envelope* $\max(f,g)$ of f and g equals the pointwise maximum of both functions. Finally, we denote by $f \circ g$ the *composition* of two functions f and g, that is, $(f \circ g)(x) := f(g(x))$ for all $x \in X$. For any $x \in \mathbb{R}$, we define $x + \infty := \infty$ and $x - \infty := -\infty$.

We define the *slope* f' of a function f at some $x' \in X$ as the corresponding *right derivative* $f'(x') := (\partial f(x)/\partial x)(x')$. (Thereby, slope is well-defined also for piecewise-defined functions.) A function f is *(monotonically) increasing* on some interval $I \subseteq X$ if for all $x_1 \in I$ and $x_2 \in I$ with $x_1 < x_2$, we get $f(x_1) \leq f(x_2)$. Conversely, if $f(x_1) \geq f(x_2)$ always holds, f is *(monotonically) decreasing* on this interval. The function f is *convex* on the interval I in case that for all $x_1 \in I$, $x_2 \in I$ with $x_1 \neq x_2$ and $\lambda \in [0,1]$, it holds that $f(\lambda x_1 + (1 - \lambda)x_2) \leq \lambda f(x_1) + (1 - \lambda)f(x_2)$. It is *concave* on I if for all values $x_1 \in I$, $x_2 \in I$ with $x_1 \neq x_2$ and $\lambda \in [0,1]$, the inequality $f(\lambda x_1 + (1 - \lambda)x_2) \geq \lambda f(x_1) + (1 - \lambda)f(x_2)$ holds. Intuitively, a function is convex (concave) on some interval if a straight line between any two points of the plotted function in this interval lies above (below) the function. For example, the function shown in Figure 3.1 is convex on the interval $[2,5)$ and concave on the interval $[5,\infty)$. A function f is called *strictly* increasing, decreasing, convex, or concave if equality is ruled out in the respective inequalities above.

Shortest Paths. An s–t *path* P, sometimes written $P_{s,t}$, is defined as a sequence $P = [s = v_1, v_2, \ldots, v_k = t]$ of vertices in a graph $G = (V,E)$, such that $(v_i, v_{i+1}) \in E$ for each $i \in \{1, \ldots, k-1\}$. If $k > 1$ and the vertices s and t coincide, we call P a *cycle*. Any path $P' = [v_i, v_{i+1}, \ldots, v_j]$ with $1 \leq i \leq j \leq k$ is called a *subpath* of P. If $i = 1$, we say that P' is a *prefix* of P. If $j = k$, P' is a *suffix* of P. Given two paths $P = [v_1, \ldots, v_i]$ and $Q = [v_i, \ldots, v_k]$, we denote by $P \circ Q := [v_1, \ldots, v_i, \ldots, v_k]$ their concatenation.

Consider a function $z \colon E \to \mathbb{R}$ mapping edges to scalar costs. The *length* or *cost* $z(P) = \sum_{i=1}^{k-1} z(v_i, v_{i+1})$ of a path P is the sum of its edge costs. (For a vertex $v \in V$, $[v]$ is a v-v path with cost 0.) For two vertices $s \in V$ and $t \in V$, an s–t path P is a *shortest path* from s to t with respect to the cost function z if $z(P)$ is minimal among all paths from s to t in G. Observe that a shortest s–t path may not exist if the graph contains a cycle of negative cost: Repeatedly including such a cycle in an s–t path, we could decrease its cost beyond any negative value. Hence, we assume in this work that graphs contain no such *negative cycles*. The *distance* $\mathrm{dist}_z(s,t)$ from s to t (with respect to the cost function z) is then defined as the cost of a shortest s–t path if it exists, otherwise $\text{dist}_z(s,t) = \infty$. Note that in general, $\text{dist}_z(s,t) \neq \text{dist}_z(t,s)$. Unless mentioned otherwise, we also presume that graphs are *(strongly) connected*, i. e., there exists an s–t path for each pair $s \in V$, $t \in V$ and therefore $\text{dist}_z(s,t) \neq \infty$. The *diameter* of a graph is the maximum length of any shortest path in the graph. Note that in cases where no cost function is specified, any of the above concepts apply if we assume uniform costs $z \equiv 1$.

Special Graphs. For a given graph $G = (V,E)$, the graph $G' = (V',E')$ is a *subgraph* of G, denoted $G' \subseteq G$, if $V' \subseteq V$ and $E' \subseteq E$. Given a subset $V' \subseteq V$ of the vertices in G, its *induced subgraph* $G' = \{V', \{(u,v) \in E \mid u \in V', v \in V'\}\}$ consists of the vertices in V' and all edges in E connecting two vertices in V'. A subgraph G' is also called a *(strongly) connected component* of G if it is a *maximal* strongly connected induced subgraph of G, i. e., there exists no induced subgraph G'' of G that is strongly connected and it holds that $G' \subsetneq G''$.

The *backward graph* $\bar{G}$ of a directed graph G is defined as $\bar{G} = (V,\bar{E})$, with the reverse edges $\bar{E} = \{(v,u) \mid (u,v) \in E\}$. A cost function z defined on G carries over to its backward graph canonically, by defining $\bar{z}(v,u) := z(u,v)$ for all $(u,v) \in E$.

Given a graph $G = (V,E)$ and a cost function z on G, an *overlay graph* of G (with respect to z) is a graph $G' = (V',E')$ with a cost function z', such that $V' \subseteq V$ and distances are preserved, i. e., $\text{dist}_{z'}(s,t) = \text{dist}_z(s,t)$ for all $s \in V'$ and $t \in V'$.

A graph $G = (V,E)$ is called *bipartite* if it contains no cycle of odd length (assuming uniform costs of 1 for every edge). A graph is called *directed acyclic graph (DAG)* if it contains no cycles at all. (Observe that undirected graphs always contain cycles unless $E = \emptyset$.) Further, an undirected graph $T = (V,E)$ is a *tree* if it is connected and $m = n - 1$. A directed graph $T = (V,E)$ is a tree with *root* $s \in V$ if $m = n - 1$ and there is an s–v path for every $v \in V$. Note that by definition, directed trees are DAGs. A subgraph $T = (V,E')$ of some graph $G = (V,E)$ is called *shortest-path tree* (with respect to a cost function z) with root $s \in V$ if it is a (directed) tree and for every $v \in V$, the s–v path in T is a shortest s–v path in G.

Partitions. A *(vertex) partition* of a graph $G = (V,E)$ is a set $\mathcal{V} = \{V_1, \ldots, V_k\}$ of *cells* $V_i \subseteq V$, with $i \in \{1, \ldots, k\}$, such that each vertex $v \in V$ is contained in exactly one cell V_i, i. e., $V_i \cap V_j = \emptyset$ for $i \neq j$ and $\bigcup_{i=1}^{k} V_i = V$. We call the subgraph of G induced by some cell V_i the *cell-induced subgraph* of V_i.

A *(nested) multilevel partition* with $L \in \mathbb{N}$ levels is a family $\Pi = \{\mathcal{V}^1, \ldots, \mathcal{V}^L\}$ of partitions with nested cells, i. e., for each level $\ell \in \{1, \ldots, L-1\}$ and cell $V_i^\ell \in \mathcal{V}^\ell$, there is a cell $V_j^{\ell+1} \in \mathcal{V}^{\ell+1}$ at level $\ell + 1$ with $V_i^\ell \subseteq V_j^{\ell+1}$. We call $V_j^{\ell+1}$ the *supercell* of V_i^ℓ. For consistency, we define $\mathcal{V}^0 := \{\{v\} \mid v \in V\}$ (the trivial partition where each vertex defines its own cell) and $\mathcal{V}^{L+1} := \{V\}$ (the trivial single-cell partition). An edge

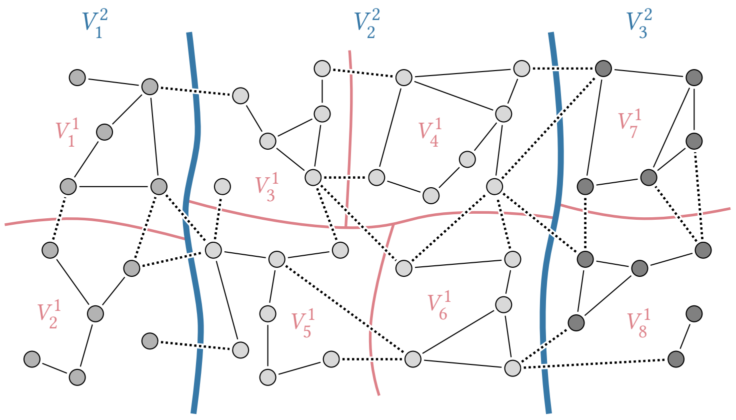


Figure 3.2: Illustration of a graph and a multilevel partition $\Pi = \{\mathcal{V}^1, \mathcal{V}^2\}$ of its vertices with two levels and the nested cells $\mathcal{V}^1 = \{V_1^1, \dots, V_8^1\}$ and $\mathcal{V}^2 = \{V_1^2, V_2^2, V_3^2\}$. Boundary edges are drawn dotted. Vertices filled in the same tone belong to the same level-2 cell.

$(u, v) \in E$ is a *boundary edge* (u and v are *boundary vertices*) on level $\ell \in \{1, \dots, L\}$ if u and v are in different cells of $\mathcal{V}^\ell$. Note that a boundary vertex on level ℓ is also a boundary vertex on lower levels. Figure 3.2 shows a multilevel partition of a graph.

Similar to vertex partitions, we use *edge partitions* $\mathcal{E} = \{E_1, \dots, E_k\}$ with $E_i \cap E_j = \emptyset$ for $i \neq j$ and $\bigcup_{i=1}^k E_i = E$. A vertex $v \in V$ is *distinct* (with respect to $\mathcal{E}$) if all its incident edges belong to the same cell, otherwise v is a *boundary vertex* or *ambiguous*.

3.2 Geometry

A *point* $p = (x, y) \in \mathbb{R}^2$ is defined by its real-valued coordinates in the Euclidean plane. We denote by pq the *line segment* between two *endpoints* $p \in \mathbb{R}^2$ and $q \in \mathbb{R}^2$, where $p \neq q$. A line segment *contains* all points that lie on it. More formally, a *closed* segment pq contains all points $p + \lambda(q - p)$ for $\lambda \in [0, 1]$, where p and q are interpreted as vectors on the vector space formed by $\mathbb{R}^2$. In particular, a closed line segment pq also contains its endpoints p and q. Conversely, an *open* line segment does not contain p and q, so $\lambda \in (0, 1)$ holds in the above term. The points p and q also define a *line* ℓ passing through them, which contains all points $p + \lambda(q - p)$ with $\lambda \in \mathbb{R}$.

Two lines or line segments p_1q_1 and p_2q_2 *intersect* if there is at least one point that is contained in both lines or line segments, respectively. Two line segments *cross* each other if they intersect in exactly one point $p \in \mathbb{R}^2$ that is no endpoint of either segment, i. e., p is distinct and $p \notin \{p_1, q_1, p_2, q_2\}$.

Polygons. A *polygonal path* or *chain* π is a sequence of line segments represented by a sequence $[p_1, \dots, p_k]$ of *endpoints*, such that exactly the (closed) line segments p_ip_{i+1}

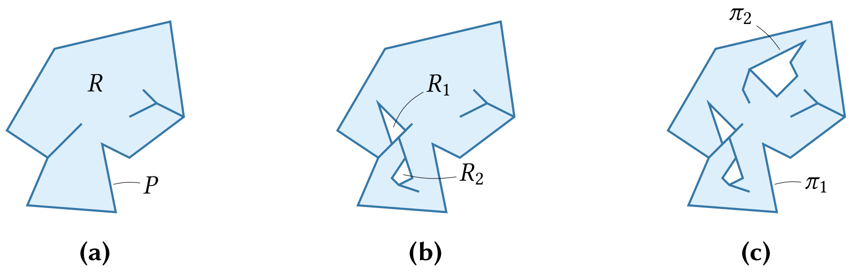


Figure 3.3: Different types of polygons. (a) A simple polygon P and its (shaded) interior R. (b) A hole-free polygon with bounded regions R_1 and R_2 enclosed by its (shaded) interior. (c) A plane polygon defined by two closed chains π_1 and π_2.

are contained in π for all $i \in \{1, \ldots, k-1\}$. We also allow the case $k = 1$, i. e., paths that consist of a single point $p \in \mathbb{R}^2$. A polygonal path is *simple* if no two line segments of the path cross each other. Note that we allow collinear segments to overlap, though. A polygonal path is *closed* if its first and last point coincide, i. e., $p_1 = p_k$. The *complexity* $|\pi|$ of a polygonal path π is the number $k-1$ of segments it is composed of.

A *polygon* is a non-empty set $P = \{\pi_1, \ldots, \pi_\ell\}$ of closed polygonal paths. Each polygonal path π_i, with $i \in \{1, \ldots, \ell\}$, is called a *(connected) component* of P. An endpoint of π_i is also called *vertex* of P and a segment between two consecutive endpoints in π_i is called *edge* of P. The *complexity* $|P|$ of a polygon P is its number of edges, i. e., the sum of the complexities of all its polygonal paths. A polygon is called *plane* or *non-crossing* if all its components are simple and any pair of components π_i and π_j with $i \neq j$ has empty intersection. Additionally, we demand that in the subdivision of the Euclidean plane defined by a plane polygon P, there is a (unique) bounded region R that intersects all points of every line segment in P; see also Figure 3.3. The set P is called the *boundary* of R. The region R bar its boundary P is called the *interior* of P. Observe that for any pair of points $p \in \mathbb{R}^2$ and $q \in \mathbb{R}^2$ in the interior of a plane polygon P, there is a polygonal path connecting p and q that is entirely contained in the interior of P. Conversely, a polygon is called *self-intersecting* if two of its components intersect or at least one component is not simple.

Geometric Graphs. In a *geometric graph* $G = (V, E)$, each vertex $v \in V$ corresponds to a (distinct) point $p_v \in \mathbb{R}^2$ in the Euclidean plane. Moreover, every edge $(u, v) \in E$ corresponds to a line segment $p_u p_v$. A geometric graph is *planar* if there exists no pair of edges (more formally, no pair of corresponding line segments) that cross each other. A planar graph subdivides the Euclidean plane into multiple regions, which are called *faces* of the graph; see Figure 3.4 for an example. The (unique) unbounded region is called the *outer face*, while all other regions are *inner faces*. A planar graph is called *outerplanar* if all points p_v corresponding to vertices $v \in V$ intersect the outer face.

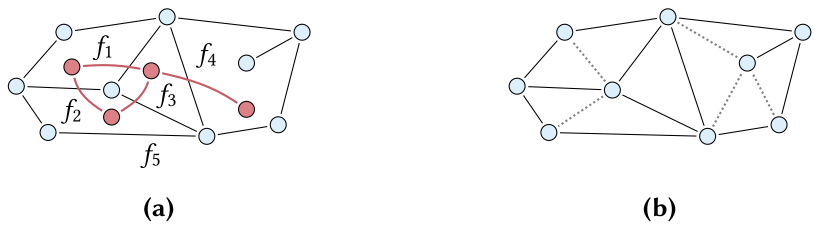


Figure 3.4: Illustration of a planar graph. (a) The graph together with its weak dual graph (red). The primal graph has the inner faces f_1, f_2, f_3, f_4 and the (unbounded) outer face f_5. (b) A triangulation of the planar graph from Figure 3.4a is obtained after adding the dotted segments.

The *(weak) dual graph* $G' = (V', E')$ of a geometric graph $G = (V, E)$ is an undirected graph that contains exactly one vertex for every inner face of G. Two vertices $u \in V'$ and $v \in V'$ are adjacent if and only if their corresponding faces in G share at least one common edge in E; see Figure 3.4a.

Every inner face f of a planar graph induces a plane polygon P_f, such that the boundary of P_f corresponds to the edges bounding the face f. Conversely, any plane polygon P induces an undirected planar graph G_P, where the vertices of G_P are the vertices of P and the edges in G_P correspond to edges in P.

Triangulations. A *triangulation* of a plane polygon P is defined as a set T of line segments, such that (1) each line segment in T is fully contained in P and its interior, (2) both endpoints of each segment in T coincide with two vertices of P, (3) different line segments in T only intersect in at most one of their endpoints, and (4) the set is maximal, i. e., no line segment can be added without violating at least one of the previous conditions. Observe that T subdivides P into smaller subregions and each of these regions is bounded by a triangle. A triangulation T of a planar graph is the union of triangulations of all its inner faces; see Figure 3.4b. Then, the segments of T can also be interpreted as edges that are added to the graph.

Special Polygons. We distinguish three types of plane polygons, described below and illustrated in Figure 3.3, to characterize polygons that correspond to inner faces of (certain types of) planar graphs. First, a *simple polygon* P subdivides the Euclidean plane into a (single) bounded region R and an unbounded region; see Figure 3.3a. For example, inner faces of outerplanar geometric graphs induce simple polygons. Second, a *hole-free polygon* is a polygon P that subdivides the Euclidean plane into an unbounded region, a (unique) bounded region R that intersects every point of every line segment in P, and an arbitrary number of other bounded regions $R_1, \ldots, R_k$; see Figure 3.3b. An inner face of a strongly connected planar graph corresponds to a hole-free polygon. Finally, the class of plane polygons defined above corresponds to the inner faces of planar graphs (which may not be strongly connected); see Figure 3.3c.

3.3 The Shortest Path Problem

Given a directed graph $G = (V, E)$ and a cost function $z\colon E \to \mathbb{R}$ on G, a general variant of the *shortest path problem* takes as input a set $S \subseteq V$ of sources and a set $T \subseteq V$ of targets. It asks for the distance $\mathrm{dist}_z(s,t)$ and a shortest path between each $s \in S$ and each $t \in T$. Certain variants of this generic problem setting often are of particular interest in the literature:

1. The *single-pair shortest path (SPSP)* problem takes as input a source $s \in V$ and a target $t \in V$ and asks for the distance $\mathrm{dist}_z(s,t)$, i. e., $S = \{s\}$ and $T = \{t\}$.
2. In the *single-source shortest path (SSSP)* problem, a source $s \in V$ is given and distances $\mathrm{dist}_z(s,v)$ from s to all vertices $v \in V$ are required, i. e., $S = \{s\}, T = V$.
3. The *all-pairs shortest path (APSP)* problem asks for the distances $\mathrm{dist}_z(u,v)$ for every $u \in V$ and $v \in V$, i. e., $S = T = V$.

In addition to the distance, any variant may also ask for an actual shortest path between each considered pair of vertices. Next, we discuss Dijkstra's algorithm, which solves the SPSP and SSSP problem, and extensions to more general problem settings (Section 3.3.1). Afterwards, we recap speedup techniques, which apply preprocessing to improve the performance of Dijkstra's algorithm (Section 3.3.2).

3.3.1 Dijkstra's Algorithm and Generalizations

The algorithm of Dijkstra [Dij59] is a well-known approach that, in its basic variant, solves the SSSP problem on a graph $G = (V, E)$ with an associated *nonnegative* cost function $z\colon E \to \mathbb{R}_{\geq 0}$. Pseudocode of the algorithm is shown in Figure 3.5. To compute, for a given source $s \in V$, the distances $\mathrm{dist}_z(s,v)$ to all vertices $v \in V$, it maintains a tentative *distance label* $d(v)$ at each vertex. Initially, the algorithm sets $d(s) = 0$ and $d(v) = \infty$ for all $v \in V \setminus \{s\}$. The source s is also inserted into a priority queue (denoted by Q in Figure 3.5), which uses $d(\cdot)$ as its key function. In each iteration of its main loop, the algorithm then extracts a vertex $u \in V$ with minimum key $d(u)$ from the priority queue, thereby *settling* (or *scanning*) it. Next, it *scans* all outgoing edges $(u,v) \in E$ of u. If $d(u) + z(u,v) < d(v)$, i. e., the tentative distance at v can be improved via (u,v), it updates $d(v)$ accordingly and adds or updates v in the priority queue. This operation is also called *edge relaxation*. The algorithm terminates as soon as the queue runs empty.

Correctness of Dijkstra's algorithm follows from its *label-setting property*: Once a vertex $v \in V$ has been extracted from the queue, the distance label $d(v)$ is final and equals the length $\mathrm{dist}_z(s,v)$ of a shortest path from s to v. This is due to the fact that when v is settled, the key $d(v)$ (and hence, the cost of the best s–v path found so far) is minimal among any vertices in the priority queue. Since edge costs are nonnegative,

```
// initialize labels
foreach $v \in V$ do
    $d(v) \longleftarrow \infty$
$d(s) \longleftarrow 0$
$Q$.insert($s$,0)
// run main loop
while $Q$.isNotEmpty() do
    $u \longleftarrow Q$.deleteMin()
    foreach $(u,v) \in E$ do
        if $d(u) + z(u,v) < d(v)$ then
            $d(v) \longleftarrow d(u) + z(u,v)$
            $Q$.update($v$,$d(v)$)
```

Figure 3.5: Pseudocode of Dijkstra's algorithm. Given a graph $G = (V,E)$, a nonnegative cost function $z\colon E \rightarrow \mathbb{R}_{\geq 0}$, and a source $s \in V$, it computes the distances $\mathrm{dist}_z(s,v)$ for all $v \in V$.

any s–v path encountered afterwards must have cost at least $d(v)$. When solving the SPSP problem, we can exploit this observation by making use of the *stopping criterion*: As soon as the target vertex $t \in V$ is extracted from the queue, we know that a shortest s–t path has been found. Hence, the algorithm can terminate at this point. While this has no effect on the asymptotic complexity of the algorithm, the number of iterations in the main loop is reduced by about a factor of 2 on average [Bau11, Lemma 2.2].

To retrieve the actual s–t path, *parent pointers* can be added: Together with its distance label, every vertex $v \in V$ maintains its parent $p(v)$, which is updated to a vertex $u \in V$ whenever the distance label $d(v)$ is improved after scanning some edge $(u,v) \in E$. Backtracking the parent pointers from an arbitrary vertex $v \in V$ then yields the vertices of a shortest s–v path (in reverse order). Using a Fibonacci heap [FT87] to implement the priority queue, the running time of Dijkstra's algorithm is in $O(n \log n + m)$. More complex data structures [Tho04] or integral bounds on the maximum distance [Ahu+90, CGS99] yield slightly better asymptotic running times. Even though they result in a higher complexity of $O((n+m)\log n)$, generalized versions [Joh75] of binary heaps [Flo64, Wil64] tend to be faster on sparse graphs (such as road networks) in practice [CGR96].

Below, we consider two common generalizations of the SPSP (and SSSP) problem formulated above. The first considers multicriteria problems, where we deal with multiple cost functions instead of a single one. The second generalizes (scalar) costs at edges to functions of some variable.

Multicriteria Shortest Paths. In multicriteria scenarios, the input graph G has more than one cost function associated with its edges. For example, there may be a

```
// initialize label sets
foreach $v \in V$ do
    $L(v) \longleftarrow \emptyset$
$L(s) \longleftarrow \{(0,...,0)\}$
$Q$.insert($(0,...,0), s, \mathrm{key}((0,...,0))$)
// run main loop
while $Q$.isNotEmpty() do
    $(\ell = (x_1,...,x_k), u) \longleftarrow Q$.deleteMin()
    foreach $(u,v) \in E$ do
        $\ell' \longleftarrow (x_1 + z_1(u,v),...,x_k + z_k(u,v))$
        if $L(v)$ does not dominate $\ell'$ then
            $L(v)$.deleteLabelsDominatedBy($\ell'$)
            $L(v)$.insert($\ell'$)
            $Q$.update($\ell', v, \mathrm{key}(\ell')$)
```

Figure 3.6: Pseudocode of the multicriteria shortest path algorithm. For a graph $G = (V,E)$, cost functions $z_1 \colon E \to \mathbb{R}_{\geq 0},...,z_k \colon E \to \mathbb{R}_{\geq 0}$, and a source $s \in V$, it computes, for each vertex $v \in V$, a maximal Pareto set of solutions corresponding to nondominated s–v paths.

cost function modeling travel time along road segments and another cost function modeling energy consumption. In a general setting, we are given $k \in \mathbb{N}$ scalar, nonnegative cost functions $z_1,...,z_k$ on G. For two vertices $s \in V$ and $t \in V$, the cost of an s–t path P can be expressed by a k-tuple $d = (z_1(P),...,z_k(P))$. In general, there is no unique path that is the shortest with respect to *every* cost function. We adopt the notion of *Pareto dominance* to compare tuples representing costs of paths: A tuple $d_1 = (x_1,...,x_k)$ *dominates* a tuple $d_2 = (y_1,...,y_k)$ if d_1 is at least as good as d_2 in any criterion. Formally, d_1 dominates d_2 if $x_i \leq y_i$ holds for all $i \in \{1,...,k\}$. A set $D = \{d_1,...,d_\ell\}$ of tuples *dominates* a tuple d if at least one tuple d_i, with $i \in \{1,...,\ell\}$, dominates d. The set D is called a *Pareto set* if there are no two tuples $d_i \in D$ and $d_j \in D$ such that $i \neq j$ and d_i dominates d_j. We extend the notion of dominance to paths as well, saying that for some $s \in V$ and $t \in V$, an s–t path P *dominates* another s–t path Q if its associated tuple dominates the one associated with Q.

The *multicriteria shortest path* algorithm [Han80, Mar84] is a natural extension of Dijkstra's algorithm to the multicriteria setting; see Figure 3.6 for pseudocode. Given a source $s \in V$, it computes, for each vertex $v \in V$, a *maximal* Pareto set of s–v paths and the associated cost tuples, i. e., any s–v path in the graph is dominated by this set. It is well known that the size of these Pareto sets can be exponential in the input size [Han80, Theorem 1]. Consequently, the worst-case running time of the search is exponential as well. Instead of scalar values, it uses *label sets* $L(\cdot)$ at vertices, which may hold several *labels* $\ell = (x_1,...,x_k)$. The algorithm starts with empty label sets at every vertex, initially adding only the label $(0,\dots,0)$ to $L(s)$ and a priority queue. Then, it works similar to Dijkstra's algorithm. In each step, it extracts a label ℓ that has the smallest associated key from the priority queue. Given the vertex $u \in V$ the label ℓ belongs to, all outgoing edges $(u,v) \in E$ are scanned. For each edge, the algorithm generates a new label ℓ' by adding the costs of (u,v) to ℓ. If ℓ' is not dominated by $L(v)$, the algorithm adds ℓ' to $L(v)$, removing labels dominated by ℓ' from $L(v)$ on-the-fly. Thereby, it maintains the invariant that $L(v)$ is a Pareto set.

The priority of labels in the queue is determined by the function $\mathrm{key}(\cdot)$. Typically, it reflects a lexicographic order of labels, though other expansion strategies are possible, too [RE09, Skr00]. For nonnegative edge costs, such key functions can ensure that the algorithm is *label setting*, that is, after a label has been extracted from the queue, it will not be dominated by any label generated later on. If only the Pareto set for a single target vertex is required, the practical performance of the algorithm can be improved: Using *target pruning* [DMS08], labels at any vertex are discarded if they are dominated by the label set at the target. To retrieve the actual path corresponding to some label, the algorithm can use *parent pointers*, storing for each label its predecessor. Observe that in the special case that only a single cost function is given ($k = 1$), the multicriteria shortest path algorithm behaves like Dijkstra's algorithm.

Profile Search. In another generalized problem setting, scalar edge costs are replaced by functions of some variable, which represents a certain state at the tail vertex of an edge [DW09, SL15]. Time-dependent route planning is probably the most well-known example, where edge costs are travel times that depend on the current point in time to account for, e. g., peak and off-peak hours [Bat+13, DW09, FHS14]. Generally, we are given a graph $G = (V,E)$ and a cost function $z\colon E \to \mathbb{F}$, associating with every edge a function from some function space $\mathbb{F}$, mapping any state at the tail of an edge to the resulting cost or state after traversing the edge. The *shortest profile problem* is a generalization of the SPSP problem that, given a source $s \in V$ and target $t \in V$, asks for an *s–t profile*, i. e., a functional representation of the distance from s to t for *every* initial state at the source (e. g., all possible departure times).

An approach based on Dijkstra's algorithm to compute an s–t profile from a given source $s \in V$ to all vertices $v \in V$ is *(label-correcting) profile search* [Dea99, OR91]. It is outlined in Figure 3.7. The algorithm requires two basic operations to generalize edge relaxation. First, the *link* operation concatenates two functions $f \in \mathbb{F}$ and $g \in \mathbb{F}$ by computing a third function $h \in \mathbb{F}$ that represents, for each state, the result of applying f and g in this order. For example, if the state is the departure time at the tail of an edge and functions map this state to the corresponding arrival time at the head, the link operation yields $\mathrm{link}(f,g) := g \circ f$. Let us assume that functions reflect costs of edges, in which case departure time is mapped to travel time on an edge and we get $\mathrm{link}(f,g) := f + g \circ (\mathrm{id} + f)$. Second, the *merge* operation takes two functions $f \in \mathbb{F}$,

```
// initialize labels
foreach $v \in V$ do
    $f_v \longleftarrow f_\infty$
$f_s \longleftarrow f_0$
$Q$.insert($s$,key($f_0$))
// run main loop
while $Q$.isNotEmpty() do
    $u \longleftarrow Q$.deleteMin()
    foreach $(u,v) \in E$ do
        $f \longleftarrow \text{link}(f_u, z(u,v))$
        if $\exists x \in X\colon f(x) < f_v(x)$ then
            $f_v \longleftarrow \text{merge}(f_v, f)$
            $Q$.update($v$,key($f_v$))
```

Figure 3.7: Pseudocode of label-correcting profile search. It requires a graph $G = (V, E)$, a cost function $z\colon E \to \mathbb{F}$, and a source vertex $s \in V$. The output is a cost function f_v for every vertex $v \in V$, which evaluates to the cost of an optimal s–v path for every initial state at s.

$g \in \mathbb{F}$ and yields the pointwise minimum $\text{merge}(f, g) := \min(f, g)$, i. e., the result is a function $h \in \mathbb{F}$ with $h(x) = \min\{f(x), g(x)\}$ for arbitrary x within the function domain. For correctness, the function space $\mathbb{F}$ must be closed under linking and merging, i. e., linking or merging two functions in $\mathbb{F}$ results in a function that is contained in $\mathbb{F}$.

In contrast to Dijkstra's algorithm, the label at a vertex $v \in V$ consists of a tentative *function* $f_v \in \mathbb{F}$ in a profile search. The algorithm initializes every label with the constant function $f_\infty \equiv \infty$, except for the source label, which is initialized with a function that always evaluates to 0. The main loop of the search works along the lines of Dijkstra's algorithm. When scanning an edge $(u, v) \in E$, instead of adding a scalar cost value to the current label, the corresponding functions f_u at u and the function $z(u, v)$ are linked. If the resulting function yields an improvement to the function at v, both functions are merged and the result is written into the label at v. The key of v in the priority queue is updated accordingly.

Profile search is *label correcting* in general, i. e., the label of some vertex is not necessarily final when it is extracted from the queue. Hence, a vertex may be reinserted and extracted from the queue more than once, but the algorithm terminates as long as edge costs are nonnegative (for arbitrary state at their tail). In case that only the s–t profile for a given target $t \in V$ is required, using $f^{\min}$ as key in the priority queue for an arbitrary $f \in \mathbb{F}$ yields a relaxed stopping criterion: After extracting a vertex $v \in V$, we check whether $f_v^{\min} \geq f_t^{\max}$, i. e., the minimum of the tentative function at v exceeds (or equals) the maximum of the function at the target t. In this case, the search can safely terminate [DW09]. To retrieve shortest paths for all different initial states, tentative functions can be enriched with parent pointers, assigning the respective preceding label with each of its (sub)functions. The performance of profile search heavily depends on the complexity of the cost functions [Dea99, OR91]. The algorithm is identical to Dijkstra's algorithm if all functions are constant and the link operation computes a constant function that equals the sum of its input functions, i. e., $\text{link}(f_a, f_b) = f_{a+b}$ for constants $a \in \mathbb{R}_{\geq 0}$ and $b \in \mathbb{R}_{\geq 0}$.

3.3.2 Speedup Techniques

Dijkstra's algorithm has low asymptotic complexity. However, even on modern hardware it may take seconds to compute shortest paths on large, realistic instances. To enable faster computation for, e. g., interactive applications, *speedup techniques* introduce a two-phase workflow. Exploiting that the topology of a road network rarely changes in practice, these techniques distinguish an offline *preprocessing phase* and an online *query phase*. Additional information on the (static) graph $G = (V, E)$ is gathered during preprocessing. This information is used to speed up shortest path computation in the query phase. Since the graph is fixed in this scenario, a query is defined solely by the source and the target. Therefore, we speak of *one-to-one* or *point-to-point* queries when the SPSP problem has to be solved. Similarly, *one-to-all* queries solve the SSSP problem, *one-to-many* queries ask for distances (or shortest paths) between one source and multiple targets, and *many-to-many* queries ask for distances (or shortest paths) between sets of sources and targets. Below, we review speedup techniques designed for point-to-point queries, namely, the A* algorithm, Contraction Hierarchies (CH), and Multilevel Dijkstra (MLD). We also describe adaptations of these techniques to more complex query scenarios.

A* Search. The A* algorithm [HNR68] is a simple extension of Dijkstra's algorithm that attempts to *guide* the search towards the target. It does so by making use of a *potential function* $\pi \colon V \to \mathbb{R}$ on the vertices [Joh77]. The potential function is called *consistent* with respect to a cost function $z \colon E \to \mathbb{R}$ if $z(u, v) - \pi(u) + \pi(v) \geq 0$ for all $(u, v) \in E$. Any consistent potential induces a nonnegative *reduced edge cost function* z^* after *shifting* the cost of every edge $(u, v) \in E$ by its potential difference, setting $z^*(u, v) := z(u, v) - \pi(u) + \pi(v)$.

The search is similar to Dijkstra's algorithm, except for one modification: The key of every vertex (in the priority queue) is increased by its potential. Using consistent potentials, the label setting property of the algorithm is maintained. This is due to the fact that running A* search on a graph with cost function z is equivalent to running Dijkstra's algorithm on the same graph, but with the reduced cost function z^* [Poh71]. An alternative explanation is that consistent potentials ensure that the minimum key in the priority queue is nondecreasing throughout the search. To see this, observe that scanning an edge $(u, v) \in E$ after extracting the vertex u with the key $d(u) + \pi(u)$ from

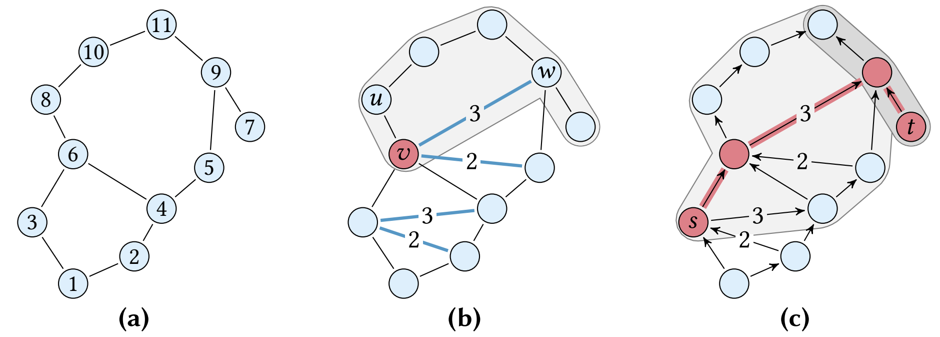


Figure 3.8: Illustration of CH preprocessing and the query phase. (a) The original (undirected) graph with uniform edge costs. Indicated vertex ranks are assigned to the input graph from bottom to top, ranging from 1 to 11. (b) The graph augmented with shortcuts (blue edges) after preprocessing. Labels at shortcut edges correspond to their costs. The shaded area indicates the overlay graph G' before contracting the vertex v (red). Contracting this vertex yields a shortcut candidate $\{u, w\}$ with cost 4. Observe that it is not inserted, because there is a shorter path from u to w with length 3 in G'. (c) Illustration of the search graphs after preprocessing. Shaded areas indicate search spaces for a query from s to t. A path of length 5 is found (red), which equals the distance between s and t in the original graph.

the queue may result in a new label with key $d(u) + z(u,v) + \pi(v)$. Due to potential consistency, this new key cannot be smaller than the previous minimum $d(u) + \pi(u)$. The fact that keys of subsequently extracted vertices are nondecreasing implies that the distance label of some vertex can never be improved after it was extracted from the queue (as this would mean that its key decreases, too). Hence, the label is final at this point and equals the distance from the source (plus a constant potential). Therefore, the algorithm may stop as soon as it settles the target vertex.

To make the search goal directed, the A* algorithm uses consistent potentials that are *lower bounds* on the remaining distance from some vertex to the target. Thereby, vertices close to the target are scanned earlier. For example, if edge costs correspond to travel time, the Euclidean distance between a vertex and the target divided by maximum travel speed in the network yields a lower bound on the remaining travel time and a consistent potential. If coordinates of vertices are given, A* search requires no preprocessing in this case [GH05, Poh71, SV86]. On real-world instances, however, the A* variant ALT [EP13, GH05, GW05] produces better potentials by utilizing distances to preselected *landmark* vertices computed during offline preprocessing.

Contraction Hierarchies. In CH [Gei+12b], queries are accelerated by augmenting the graph with *shortcut edges*. These shortcuts are computed during a preprocessing routine that iteratively *contracts* all vertices of the input graph in a certain (heuristic) order rank: $V \rightarrow \{1, \ldots, n\}$; see Figure 3.8a. To this end, an overlay graph G' is maintained, initially set to $G' = G$. When a vertex $v \in V$ is contracted, it is removed from G'

together with all its incident edges. Shortcut edges are added between its uncontracted neighbors to preserve distances (with respect to a given cost function $z\colon E \to \mathbb{R}_{\geq 0}$) in the overlay graph G', if necessary. Let $u \in V$ and $w \in V$ be neighbors of v in G'. To determine whether a shortcut candidate (u, v) is needed, a *witness search* is run: Before adding the shortcut (u, w) with cost $z(u, v) + z(v, w)$ to G', Dijkstra's algorithm computes the distance $\mathrm{dist}_z(u, w)$ in G' (after removing v, but before adding the shortcut). If $\mathrm{dist}_z(u, w)$ is at most the cost of the shortcut candidate, the shortcut is not inserted, as it is not required to preserve distances in G' after contracting v; see Figure 3.8b. Eventually, contracting all vertices results in an “empty” graph $G' = (\emptyset, \emptyset)$.

The result of the preprocessing phase is the contraction order $\mathrm{rank}(\cdot)$ itself and the set E^+ of all shortcuts that were inserted into the overlay. They are used to create search graphs in the query phase; see Figure 3.8c. Given a source $s \in V$ and a target $t \in V$, the query algorithm is bidirectional, with a *forward search* from $s \in V$ on the graph $G^{\uparrow} := (V, E^{\uparrow})$, where $E^{\uparrow} := \{(u, v) \in E \cup E^+ \colon \mathrm{rank}(u) < \mathrm{rank}(v)\}$ is the set of *upward edges* with respect to the contraction order. The *backward search* from $t \in V$ operates on the backward graph $\bar{G}^{\downarrow}$ of $G^{\downarrow} := (V, E^{\downarrow})$, where the set of *downward edges* is given as $E^{\downarrow} := \{(u, v) \in E \cup E^+ \colon \mathrm{rank}(u) > \mathrm{rank}(v)\}$. In other words, both searches follow only paths of increasing rank. One can show that this bidirectional variant of Dijkstra's algorithm computes the distance between s and t with respect to the input graph [Gei+12b]. To retrieve the shortest path itself, shortcut edges store a pointer to the contracted *via vertex* that lead to creation of the shortcut. Recursive *path unpacking* then provides a shortest path in the input graph [Gei+12b].

Multilevel Dijkstra. The three-phase workflow of CRP [Del+17] separates preprocessing into *metric-independent preprocessing* and metric-dependent *customization*. Upon changes in the cost function due to, e. g., traffic updates or user preferences, only the customization phase has to be repeated. Delling et al. [Del+17] propose MLD to implement this generic workflow. The basic idea of this approach is to compute an overlay induced by a multilevel partition and use shortcuts to skip large parts of the original graph during queries. Customization is fast, because it only requires that the *costs* of these shortcuts are recomputed by local searches.

During (metric-independent) preprocessing, a multilevel partition Π with $L \in \mathbb{N}$ levels of the vertices in the input graph is computed. For each level $\ell \in \{1, \ldots, L\}$, this induces an overlay graph H^{ℓ} that consists of all boundary vertices and boundary edges in the partition $\mathcal{V}^{\ell}$ at level ℓ, plus cliques of internal shortcut edges connecting each pair of boundary vertices that belongs to the same cell at level ℓ. The costs of all shortcuts are computed in the customization phase by running, e. g., Dijkstra's algorithm on the respective cell-induced subgraphs. For fast integration of new cost functions, previously computed low-level overlays are used to compute shortcuts on higher levels. Finally, to answer s–t queries, Dijkstra's algorithm is run on the union of

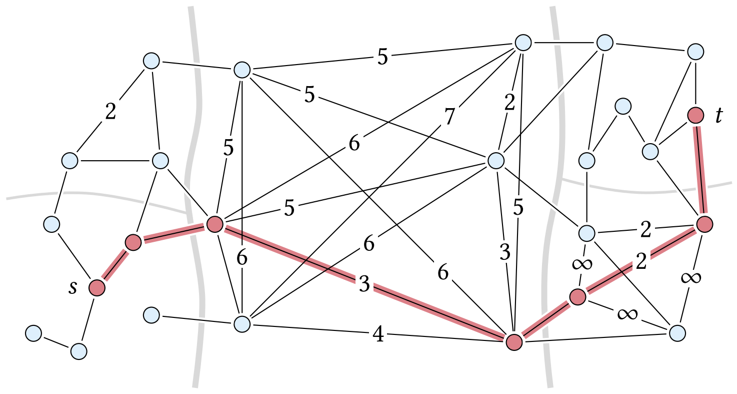
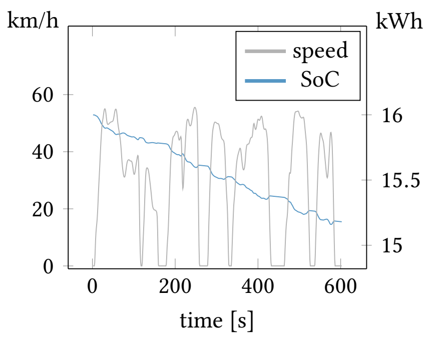


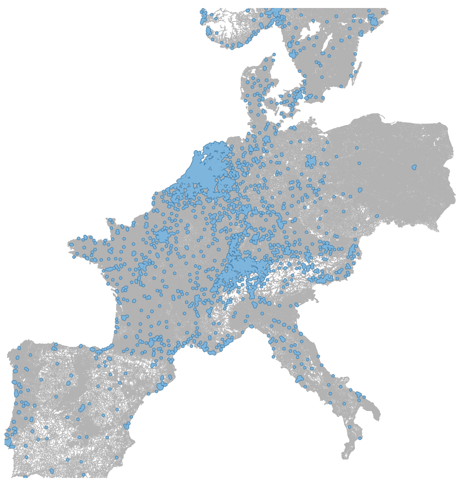
Figure 3.9: Search graph of an MLD query from s to t, induced by the multilevel partition shown in Figure 3.2. Assuming uniform edge costs of 1 in the original graph, shortcut labels indicate costs if they are greater than 1. Note that shortcuts with infinite cost may occur if a cell-induced subgraph is not strongly connected. The shortest path (red) has length 9, as in the original graph; c. f. Figure 3.2.

the top-level overlay H^L and the subgraphs of $H^{\ell-1}$ induced by the cells V_i^ℓ containing $s \in V$ or $t \in V$ on each level $\ell \in \{1,\dots,L\}$, with $H^0 = G$. This yields the correct distance from s to t with respect to the original graph [Del+17]; see Figure 3.9 for an example. To retrieve the actual path, shortcuts are unpacked recursively by running Dijkstra's algorithm between endpoints of shortcuts on the cell-induced subgraphs at the level below, until the corresponding subpath in the original graph is found.

Batched Shortest Paths. Given a source vertex $s \in V$, *one-to-all* and *one-to-many* queries ask for distances from s to all $v \in V$ or to all $t \in T$ from a given subset $T \subseteq V$, respectively. Both CH and MLD compute distances between pairs of vertices in their basic form, but can be extended to handle such batched shortest path settings.

PHAST [Del+13b] leverages CH preprocessing for fast one-to-all queries. In addition to preprocessing of plain CH, it assigns *levels* $\ell(\cdot)$ to vertices during preprocessing, initially set to 0. When contracting a vertex $u \in V$, it sets $\ell(v) = \max\{\ell(v), \ell(u) + 1\}$ for each uncontracted neighbor v of u. To answer one-to-all queries, the *upward phase* runs a forward CH search from the source. Afterwards, the *scanning phase* processes vertices in descending order of level and propagates distances by scanning, for each vertex $v \in V$, its incoming edges (u,v) in $E^\downarrow$. The algorithm exploits that vertices are represented by indices $\{1,\dots,n\}$ in practice: Since the instruction flow of the scanning phase depends solely on the contraction order, vertices are reordered during preprocessing, such that vertices at higher levels are assigned lower indices. Then, the scanning phase boils down to a linear scan over a sorted edge array, making PHAST an order of magnitude faster than Dijkstra's algorithm.





(a) (b)

Figure 3.10: Input data for our experiments. (a) A driving cycle used by PHEM, representing a collector road with heavy traffic. The plot shows changes in driving speed and SoC over time during a ride. Observe that energy is consumed even when the vehicle is not moving (due to auxiliary consumers) and energy is recuperated when braking. (b) Our main test instance, representing the road network of Western Europe. Blue dots indicate locations of charging stations. There are clear differences in the distribution of charging stations, which is very dense in the Netherlands and Switzerland, whereas Spain, Italy, and Poland contain relatively few charging stations in our data set.

RPHAST [DGW11] is an extension of PHAST to enable fast one-to-many queries. To this end, it introduces an additional *target selection* phase after preprocessing. Given a target set $T \subseteq V$, target selection extracts a subgraph $G_T^{\downarrow}$ of the original downward graph $G^{\downarrow}$, namely, the union of the search spaces of all targets $t \in T$. The graph $G_T^{\downarrow}$ is obtained in a multi-source variant of a breadth-first search (BFS) [Cor+09] from all $t \in T$ in the backward graph of $G^{\downarrow}$. All vertices visited by the BFS and their incoming edges are added to $G_T^{\downarrow}$. Afterwards, RPHAST queries resemble PHAST queries, running the scanning phase on $G_T^{\downarrow}$ instead of $G^{\downarrow}$.

GRASP [EP14] is an adaptation of MLD to batched query types. In addition to shortcuts between boundary vertices within the same cell, each level-ℓ boundary vertex, with $\ell \in \{0, \dots, L-1\}$, stores incoming *downward shortcuts* from all boundary vertices of its corresponding supercell at level $\ell + 1$. (Recall that for $\ell = 0$, *every* vertex is a boundary vertex.) Customization works similar to MLD, storing downward edges in a separate *downward graph* $H^{\downarrow}$. For one-to-all queries, the *upward phase* runs an MLD search from the source, i. e., Dijkstra's algorithm on the union of the top-level overlay and all subgraphs of overlays induced by cells containing s. After the upward phase, all vertices settled by the algorithm have correct distance labels. Then, the *scanning phase* processes cells in descending level order, sweeping over downward edges to propagate distance labels from boundary vertices to those at the level below. The number of downward edges can be reduced via *edge reduction* [ETP12], omitting downward shortcuts (u, v) in a cell V_i^ℓ at level $\ell \in \{1, \dots, L\}$ if the shortest u–v path within the subgraph of $H^{\ell-1}$ induced by V_i^ℓ contains another boundary vertex of V_i^ℓ.

3.4 Experimental Setup

In this section, we describe the experimental setup on which our evaluations in the subsequent chapters are based. We explain in detail how realistic input data is obtained for our main benchmark instance, which represents the road network of Western Europe; see Figure 3.10. For energy consumption, we use highly detailed data measured from a real production vehicle (Peugeot iOn). Any modifications to the basic experimental setup described below are mentioned in the respective main chapters.

Methodology. We implemented all evaluated algorithms in C++, using g++ 4.8.5 (flag -O3) as compiler and OpenMP for parallelization. Obtained results were always checked against reference implementations (typically variants of Dijkstra's algorithm) for correctness. Experiments were conducted on two different machines, depending on whether the considered technique exploits parallelism. Details are given below.

- Experiments involving parallel algorithms were conducted on two 8-core Intel Xeon E5-2670 clocked at 2.6 Ghz, with 64 GiB of DDR3-1600 RAM, 20 MiB of L3 cache, and 256 KiB of L2 cache, hereafter denoted machine-p.
- All other experiments were conducted on a single core of a 4-core Intel Xeon E5-1630v3 clocked at 3.7 GHz, with 128 GiB of DDR4-2133 RAM, 10 MiB of L3 cache, and 256 KiB of L2 cache, hereafter denoted machine-s.

Main Input. Our main benchmark instance, named Eur-PTV, is based on the road network of Western Europe, kindly provided by PTV AG.[1] Road segments have associated lengths, average speeds and road categories. Travel times on road segments were directly extracted from these data. To generate energy consumption along road segments, their slopes are required, as they affect consumption. We retrieved elevation information for the vertices from the freely available NASA Shuttle Radar Topography Mission (SRTM) data set.[2] It covers large parts of the world with tiles at a resolution of three arc seconds (approximately 90 meters at the equator). The elevation of a vertex was obtained by bilinear interpolation from the four corners of the SRTM tile containing the vertex. Previous studies on the effect of elevation models on data

[1]`http://www.ptvgroup.com`
[2]`http://srtm.csi.cgiar.org`

Table 3.1: Overview of instances used in our experiments. For each instance, the table shows the underlying road network, the number of vertices, and the number of edges. Note that OSM instances have many degree-2 vertices, meant primarily for visualization.

Instance	Road Network	# Vertices	# Edges
Eur-PTV	Western Europe	22 198 628	51 088 095
Eur-DIMACS	Western Europe	18 010 173	42 188 664
Jap-OSM	Japan	25 970 678	54 141 580
Ger-OSM	Germany	23 913 390	48 239 355
Ger-PTV	Germany	4 692 091	10 805 429
Sgr-OSM	Southern Germany	5 588 146	11 711 088
Swi-OSM	Switzerland	3 259 674	6 488 514

quality indicate that bilinear interpolation (among other techniques) yields the most accurate results [GAP15]. We filled (rarely) missing data samples by interpolating from neighbors. We removed all vertices from the graph where no elevation data is available (not even via sensible interpolation), such as large parts of Scandinavia, where the data ends beyond the 60th circle of latitude.

Our energy consumption data originates from the Passenger Car and Heavy Duty Emission Model (PHEM), developed by the Graz University of Technology [Hau+09]. PHEM is a micro-scale emission model based on backwards longitudinal dynamics simulation. Besides other applications, PHEM is used to calculate emissions for passenger cars as well as heavy and light duty vehicles for the Handbook of Emission Factors for Road Transport (HBEFA) [Hau+09]. The HBEFA driving cycles cover a large variety of road categories, speed limits, traffic situations, and slopes. These cycles were calculated using different EV configurations and vehicle types to generate energy consumption estimates for all available driving situations. Figure 3.10a shows a driving cycle for a collector road under heavy traffic conditions. We carefully mapped consumption data obtained from PHEM to our network by a heuristic that measures the similarity between road segments of the network and the parameters of PHEM. We deleted edges that cannot be mapped to a PHEM road category (such as private roads and ferries). Finally, we extracted the largest strongly connected component from the remaining input. As a consequence, the United Kingdom is not contained in our instance (as it is only reachable by ferry from continental Europe in our data set). The resulting graph consists of 22 198 628 vertices and 51 088 095 edges.

In our experiments, we use different instances of the PHEM data. The first is based on a Peugeot iOn. The second is an artificial EV model [Tie+12]. Unless mentioned otherwise, we disable auxiliary consumers to get the best possible cruising range. Besides extending the range, this also increases the amount of road segments where the vehicle is able to recuperate (making the instances only “harder” for our algorithms): The resulting amount of edges with negative energy consumption is 11.8 % and 15.2 %

for the model based on a Peugeot iOn and the artificial model, respectively. Edge consumption values are stored in mWh, to avoid rounding issues along short-distance edges. In our experiments, we typically consider two realistic long-range battery capacities of 16 kWh (the maximum capacity of a Peugeot iOn, corresponding to a range of 100–150 km) and 85 kWh (similar to that of recent high-end Tesla models with a range of 400–500 km).

For the evaluation of algorithms involving charging stops, we located charging stations on ChargeMap.[3] Each station was mapped to its closest vertex in the network, except when there was no vertex within a radius of 20 meters around the station (in which case it was discarded). Figure 3.10b shows our main benchmark instance and the distribution of the 13 810 remaining charging stations (extracted in April 2015). Aside from the real-world data from ChargeMap, we also use artificial, random distributions in our experiments.

Alternative Inputs. For the purpose of comparison with related work, we also consider a simpler consumption model used in previous studies [EFS11, Sto12a, Sto13]. In this model, energy consumption of an edge $e = (u,v) \in E$ is assumed to depend only on horizontal length $\ell_e \in \mathbb{R}_{\geq 0}$ of the edge and vertical heights $h_u \in \mathbb{R}_{\geq 0}$ and $h_v \in \mathbb{R}_{\geq 0}$ of the vertices. More precisely, the energy consumption $c(e)$ of e is

$$c(e) := \begin{cases} \kappa \cdot \ell_e + \lambda \cdot (h_v - h_u) & \text{if } h_v - h_u \geq 0, \\ \kappa \cdot \ell_e + \mu \cdot (h_v - h_u) & \text{otherwise.} \end{cases} \tag{3.1}$$

In accordance with the previous studies [Sto13], we set $\kappa = 0.02$, $\lambda = 1$, and $\mu = 0.25$. Note that when applying this model our main instance Eur-PTV, we observe that the amount of negative edges drops to 4.4 %.

Besides our main benchmark instance described above, we also consider numerous other road graphs, which are listed in Table 3.1. This includes another graph based on the same data set as our main instance, which represents the road network of Germany (Ger-PTV) and is a subnetwork of Eur-PTV. We also performed experiments on an alternative instance that represents the European road network (Eur-DIMACS). Made available for the 9th DIMACS Implementation Challenge [DGJ09], it comes with travel times measured in seconds for every edge, but no energy consumption. Finally, we consider several instances extracted from OpenStreetMap (OSM).[4] For comparison, we present results on instances used in previous studies, which are OSM exports of the road networks of Southern Germany (Sgr-OSM) [Sto12a] and Japan (Jap-OSM) [Sto13], augmented with SRTM data to derive energy consumption on edges. In addition to that, we work with OSM exports of Germany (Ger-OSM) and Switzerland (Swi-OSM). Note that instances based on OSM data are notorious for having exceptionally many

[3]`http://www.chargemap.com`
[4]`http://www.openstreetmap.org`

vertices of degree 2 that (only) model geometry. This explains the relatively large size of these instances (e. g., the graphs representing Japan and Germany contain more vertices than our main benchmark instance, which corresponds to the road network of Western Europe).

Implementation Details. We provide details about basic algorithms and data structures used by the techniques that we evaluate in our experimental studies. In our implementation, graphs are stored in adjacency arrays [Cor+09], following the dynamic data structures of Delling [Del09] for efficient insertion and deletion of shortcuts during the preprocessing routine of CH. Costs of edges (reflecting length, travel time, or energy consumption) are stored as 32-bit integers at a reasonably high resolution (e. g., 1 mWh for energy consumption values). Additional flags indicate edge directions for bidirectional techniques [Del09]. Our implementation of Dijkstra's algorithm and techniques based on it use a 4-heap [Cor+09, Joh75] as priority queue. Unless mentioned otherwise, we use timestamps for (re)initialization of distance labels between subsequent queries in all techniques [Paj13]. Our multicriteria search algorithms always maintain priority queues of vertices (rather than labels). The key of a vertex in the queue corresponds to the smallest key of any of its unsettled labels. Note that this does not alter the order in which labels are scanned by the algorithm, but the number of entries in the priority queue decreases. More specific implementation details are mentioned in the respective sections of each main chapter of the thesis.

4 Energy-Optimal Routes for Battery Electric Vehicles

Route planning services explicitly designed for EVs have to address specific aspects, since EVs usually employ a rather limited cruising range. We study the problem of computing routes that minimize energy consumption, in order to maximize cruising range and for drivers to overcome range anxiety. This imposes nontrivial challenges. First of all, EVs can *recuperate* energy (e. g., when going downhill), but the *battery capacity* limits the amount of recoverable energy [EFS11, Sac+11]. As a result, the energy-optimal route depends on the initial *state of charge (SoC)*. This dependency is captured by the notion of *(consumption) profiles*, which map SoC at the source to (minimum) energy consumption that is necessary to reach the target [EFS11, SLW14]. Profiles are relevant in many applications where the SoC at the start of a journey is either unknown or can be decided by the driver, e. g., when charging overnight. Moreover, they are an important ingredient of speedup techniques, where preprocessing is applied to the input network for faster query times [EFS11].

In addition to the above issues, *recharging* en route may become inevitable on long-distance trips. Given that charging stations are scarce, such stops need to be planned in advance [SF12]. Therefore, we also discuss approaches that explicitly consider stops at charging stations. Note that, even when optimizing for energy-consumption only, the integration of charging stations into route planning is a nontrivial task: Recharging to a full battery at a station can be wasteful if it prevents the battery from recuperating energy on a downhill ride later on.

Modeling energy consumption precisely is another important aspect in route planning applications designed for EVs. Energy consumption is strongly influenced by a number of factors, such as vehicle load, auxiliary consumers, weather condition, driving style, and traffic conditions [SHS11]. While some factors are static, others, such as weather conditions and vehicle load, are not. Consequently, any kind of route planning approach must allow frequent updates of energy consumption data. Speedup techniques [Bas+16] for route planning in road networks, on the other hand, use a potentially costly preprocessing phase to accelerate Dijkstra's algorithm. Most were developed for static edge costs representing travel times. More recent *customizable* techniques, such as CRP introduced by Delling et al. [Del+17], are a notable exception. Based on multilevel overlay graphs [Del+09, HSW09, JP02, SWW00, SWZ02], CRP is designed to work with arbitrary cost functions and integrates new cost functions quickly, making it a promising candidate for our scenario.

In this chapter, we cover different algorithmic problems in the context of energy-optimal route planning for EVs. It turns out that these problem settings allow efficient solutions, not only in theory, but also in practice: Even for the most complex problems considered, our algorithms compute (empirically) optimal results for long-distance route queries in well below a second. Furthermore, many insights from this chapter are of fundamental importance for more involved scenarios, which we consider in Chapter 5. There, the additional consideration of overall trip time in route optimization yields significantly more complex problem settings.

Chapter Overview. In Section 4.1, we formally introduce our model of battery constraints and state two problem variants regarding energy-optimal routing for EVs. In particular, we examine *profiles* mapping initial SoC to energy consumption for a fixed pair of source and target. We recap how such profiles can be modeled as a special form of piecewise linear functions [EFS11]. As a main result of this section, we then prove that the number of breakpoints of such functions is linear in the worst case. Note that this stands in stark contrast to profiles in time-dependent routing, which can have superpolynomial size [FHS14]. We also derive basic operations to concatenate and merge profiles.

Using these insights, we investigate approaches based on Dijkstra's algorithm to compute energy-optimal routes and profiles in Section 4.2. Aiming at practical solutions, we explore different strategies to handle recuperation (i. e., negative costs). We also present a polynomial-time algorithm to compute profiles. Thereby, we efficiently solve a problem that is not only relevant on its own, but is a crucial ingredient of speedup techniques in our scenario.

Section 4.3 deals with the extended problem setting of energy-optimal routes that may include charging stops. First, we derive a baseline algorithm based on bicriteria search [Han80], which does not require any preprocessing. Second, we show that given the results from Section 4.1, the problem can be solved in polynomial time. Third, we propose a more practical technique based on a tuned implementation of CH for EVs [EFS11]. To this end, we carefully integrate battery constraints and profile search into preprocessing and the query algorithm. Although our practical implementation formally drops correctness, it *always* finds the optimal solution in our tests. For further speedup, we discuss combinations with variants of the A* algorithm.

In Section 4.4, we introduce an approach to optimize energy consumption of EVs that is designed to be fast both in (metric-dependent) preprocessing of the whole network as well as in answering queries. For that, we use ingredients from previous sections and extend the CRP method of Delling et al. [Del+17] to handle battery constraints and achieve fast (metric-dependent) preprocessing. We propose several query algorithms to compute energy-optimal routes.

Section 4.5 presents our experimental results. On the road network of Europe, we evaluate different strategies of the baseline approaches that adapt Dijkstra's algorithm. For routes including charging stops, we conduct experiments with our different approaches, considering various types and distributions of charging stations. We also evaluate our customizable technique and its query variants. Furthermore, we present a comparison of several approaches for computing energy-optimal routes for EVs. Compared to baseline algorithms, our more sophisticated techniques achieve speedups by three orders of magnitude, clearly outperforming previous approaches. We conclude this chapter with final remarks in Section 4.6.

4.1 Integrating Battery Constraints

In what follows, we first describe how we model energy consumption in our input. Further, we formally define two relevant problem settings in the context of energy-optimal routes for EVs (Section 4.1.1). Given a source and a target, the first asks for an energy-optimal path subject to a given initial SoC at the source. The second asks for a *profile*, i. e., an energy-optimal path for *every* possible SoC at the source. Afterwards, we examine the complexity of such profiles (Section 4.1.2). Finally, we show how profiles can be represented efficiently and introduce necessary operations to obtain a new profile from the profiles of two consecutive subpaths or two paths between the same pair of vertices (Section 4.1.3).

4.1.1 Model and Problem Statement

We model the road network as a directed graph $G = (V, E)$. Vertices have associated elevation values (relevant for energy consumption) given by a function $h\colon V \to \mathbb{R}_{\geq 0}$. We assume that the slope along an edge is constant—varying slopes can be modeled by adding intermediate vertices, so this is not a restriction in practice. The actual energy consumption of an EV when driving along an edge is given by the function $c\colon E \to \mathbb{R}$. Consumption can be negative to account for recuperation. However, cycles with negative consumption are physically ruled out. In other words, driving in a cycle never increases the SoC of an EV.

We assume that the EV is equipped with a battery of limited capacity $M \in \mathbb{R}_{\geq 0}$. Given the current SoC $b_u \in [0, M]$ of a vehicle positioned at some vertex $u \in V$ in the network, traversing an edge $(u, v) \in E$ typically results in the SoC $b_v = b_u - c(u, v)$. However, we must also take *battery constraints* into account: The SoC b_v must neither exceed the limit M nor drop below a predefined (e. g., user-specific) minimum [Art+10a, Art+10b, EFS11]. For the sake of simplicity and without loss of generality, we assume in this work that the minimum SoC is 0 and that $c(e) \in [-M, M]$ for all edges $e \in E$ of the input graph. Then, if the consumption $c(u, v)$ of an edge $(u, v) \in E$ exceeds the SoC b_u at u, the edge cannot be traversed, as the battery would run empty along the way. We indicate this case by setting $b_v := -\infty$. Conversely, if the battery is (almost) fully charged, passing an edge with negative consumption cannot increase the SoC beyond the maximum value M, so we obtain $b_v = M$. Given some initial SoC b_s at a source $s \in V$ together with a target $t \in V$, we say that an s–t path P is *feasible* if and only if the battery never runs empty, i. e., the SoC b_v obtained at every vertex v of P after iteratively applying the above constraints is in the interval $[0, M]$. Let b_t denote the SoC at the last vertex t of the path P. Then the *energy consumption* on P is the difference $b_s - b_t$ between the initial and the final SoC. Recall that this value can become negative due to recuperation or infinite if P is infeasible. Moreover, note that a path may be infeasible even if its cost (i. e., the sum of its consumption values) does not exceed b_s: Due to negative edge costs, there might be a prefix of greater total cost that renders the path infeasible.

In this chapter, we study two query types on the input graph, namely *SoC queries* and *profile queries*. In an SoC query, one is given a source $s \in V$, a target $t \in V$, and an initial SoC $b_s \in [0, M]$. It asks for a (single) *energy-optimal* s–t path when departing at s with SoC b_s, i. e., a path that maximizes the SoC b_t at t. (In Section 4.3, we slightly alter the notion of energy-optimal paths to take charging stops into account.) A profile query does not take b_s as input, but asks for an s–t *profile*, i. e., the optimal value b_t for *every* initial SoC $b_s \in [0, M]$. We will see that not only the maximum SoC at the target, but also the optimal path itself may vary for different values b_s of initial SoC. Hence, a profile corresponds to a *set* of optimal s–t paths.

Profiles are helpful for deciding how much to charge the battery before departing. Moreover, they are a preprocessing ingredient to our speedup techniques in subsequent sections. Thus, we examine the complexity of profiles, before we turn to efficient algorithms for solving both problem variants. In all algorithmic descriptions given in this chapter, we focus on computing the optimal SoC at the target, rather than explicitly constructing the corresponding s–t path. To obtain the actual path, one can apply backtracking or add parent pointers, as in Dijkstra's algorithm (see Section 3.3).

4.1.2 On the Complexity of Profiles

Apparently, the energy consumption along a certain s–t path may vary for different values of initial SoC at the source $s \in V$, due to battery constraints. We discuss how to efficiently compute and represent this correlation between initial SoC and energy consumption. It turns out that not only the SoC at the target $t \in V$, but also the optimal path itself depends on the initial SoC at the source s.

Given two vertices $s \in V$ and $t \in V$ of the input graph, we define the *SoC function* $f \colon [0, M] \cup \{-\infty\} \to [0, M] \cup \{-\infty\}$, also called *SoC profile*, to represent the s–t profile. The function f maps SoC at the source s to the *optimal* resulting SoC at the target t. Recall that $-\infty$ is a special value to represent insufficient charge, hence we define $f(-\infty) := -\infty$. For some s–t path P, we denote by f_P the profile of P, i. e., the SoC function that maps initial SoC at s to the resulting SoC at t after traversing P. Given the SoC functions f_P and f_Q of two paths P and Q, we say that f_P *dominates* f_Q

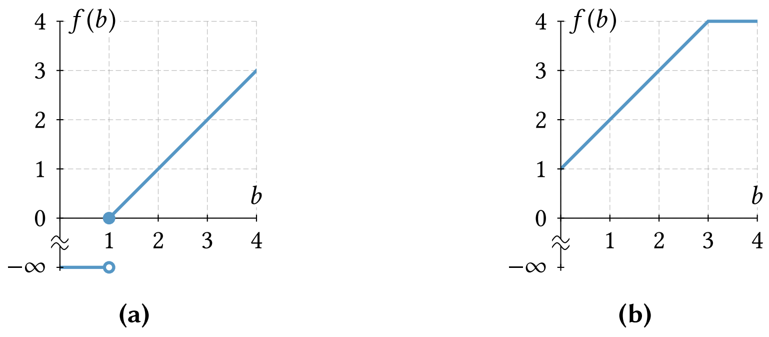


Figure 4.1: SoC functions for different edge costs, assuming a battery capacity of $M = 4$. (a) The SoC function of an edge with cost 1. (b) The SoC function of an edge with cost -1.

(similarly, P *dominates* Q) on a certain interval $I \subseteq [0, M]$ if $f_P(b) \geq f_Q(b)$ holds for all $b \in I$. If the interval is not stated explicitly, we assume $I = [0, M]$.

Below, we examine the SoC function of a given *edge* and along a fixed *path*. In both cases, the function is piecewise linear and can be represented by a constant number of breakpoints. Afterwards, we consider the general scenario, where multiple paths may contribute to the same profile. In accordance with Section 3.1, we use a sequence $F = [(x_1, y_1), \ldots, (x_k, y_k)]$ of breakpoints to define a piecewise linear SoC function f, such that $f(b) = -\infty$ for $b < x_1$, $f_P(b) = y_k$ for $b \geq x_k$, and the function is evaluated by linear interpolation for $b \in [x_1, x_k)$.

Profiles Representing Edges. We begin by describing the SoC function $f_{(u,v)}$ that reflects battery constraints for a given edge $(u, v) \in E$. We distinguish two cases. First, let the cost $c(u, v) = a^+ \geq 0$ of the edge be a *nonnegative* constant. In this case, the edge can only be traversed if the SoC at u is at least a^+. We obtain the SoC function

$$f_{(u,v)}(b) := \begin{cases} b - a^+ & \text{if } b \geq a^+, \\ -\infty & \text{otherwise.} \end{cases} \tag{4.1}$$

The function $f_{(u,v)}$ is represented by the sequence $F_{(u,v)} = [(a^+, 0), (M, M - a^+)]$ consisting of two breakpoints; see Figure 4.1a for an example. Second, if (u, v) has *negative* cost, i. e., $c(u, v) = a^- < 0$, we have to ensure that the SoC at v does not exceed the battery capacity M. We obtain the profile

$$f_{(u,v)}(b) := \begin{cases} b - a^- & \text{if } b - a^- \leq M, \\ M & \text{otherwise.} \end{cases} \tag{4.2}$$

Again, the SoC function is represented by a sequence consisting of two breakpoints, namely $F_{(u,v)} = [(0, -a^-), (M + a^-, M)]$. Figure 4.1b shows an SoC function that represents a single edge with negative cost.

Profiles Representing Paths. Eisner et al. [EFS11] show that the number of breakpoints of the SoC function f_P of a given s–t path P is bounded by a constant. For the sake of self-containedness, we give an alternative proof of this fundamental insight in Lemma 4.1. Additionally, Lemma 4.1 provides a general specification of SoC functions for single paths.

Before proving Lemma 4.1, we begin by defining *important* subpaths of an s–t path P that affect the SoC function f_P. First, let P_s^+ denote the *maximum prefix* of P, i. e., the prefix of P that has maximum cost $c(P_s^+)$ among all its prefixes. (Recall that the cost of a path is defined as the sum of its edge costs, hence, battery constraints do not apply.) If no prefix of P (including P itself) has positive cost, we obtain $P_s^+ = [s]$ and $c(P_s^+) = 0$. Similarly, the *minimum prefix* P_s^- minimizes the cost $c(P_s^-)$ among all prefixes of P. We obtain $P_s^- = [s]$ and $c(P_s^-) = 0$ in case that no prefix of P is negative. The *maximum suffix* P_t^+ and *minimum suffix* P_t^- are defined symmetrically. For the sake of simplicity, we assume in the remainder of this section that P contains no subpath with cost 0 consisting of more than one vertex (this can be enforced by perturbation of edge costs). Thus, the above subpaths are uniquely defined. Moreover, observe that $P = P_s^+ \circ P_t^- = P_s^- \circ P_t^+$; see Figure 4.2. The following Lemma 4.1 shows that the SoC function f_P (defined by its breakpoints) of a path P is completely determined by the costs of its important subpaths.

Lemma 4.1. *Given an s–t path P, its SoC function f_P is a piecewise linear function. It is defined by a sequence F_P of breakpoints in the following way.*

1. *If there exists a subpath of P with cost greater than M, $F_P = \emptyset$ and $f_P \equiv -\infty$.*
2. *Otherwise, if there is a subpath of P with cost below $-M$, $F_P = [(c(P_s^+), M - c(P_t^+))]$.*
3. *If neither such subpath exists, $F_P = [(c(P_s^+), -c(P_t^-)), (M + c(P_s^-), M - c(P_t^+))]$.*

Proof. To prove the claim, we consider the three cases separately. For each, we examine certain subpaths of P. A subpath denoted $P_{u,v}$ starts at the vertex $u \in V$ and ends at the vertex $v \in V$.

Case 1: There exists a subpath $P_{u,v}$ of P such that $c(P_{u,v}) > M$. Regardless of the SoC at u, the u–v subpath cannot be traversed. Hence, the path P is infeasible for arbitrary initial SoC and we obtain the SoC function $f_P \equiv -\infty$.

Case 2: No subpath of P has cost greater than M, but there exists a subpath $P_{u,v}$ such that $c(P_{u,v}) < -M$. Without loss of generality, let $P_{u,v}$ be the minimum-cost subpath of P, i. e., any subpath of P has cost at least $c(P_{u,v})$. We can separate P into three subpaths, namely, a prefix $P_{s,u}$, the negative subpath $P_{u,v}$, and a suffix $P_{v,t}$.

We claim that $P_{s,u}$ is in fact the maximum prefix of P. Assume for contradiction that the maximum prefix $P_{s,w}$ ends at some vertex $w \neq u$. We distinguish three cases. First, assume that w lies on the subpath $P_{s,u}$. Then the subpath $P_{w,u}$ from w to u has negative cost, because the prefix $P_{s,w} \circ P_{w,u}$ must have lower cost than the maximum

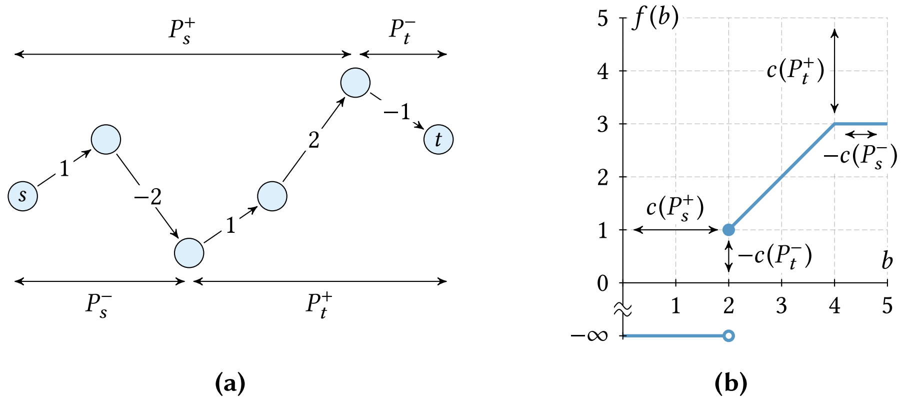


Figure 4.2: An s–t path together with its SoC function, assuming that the battery capacity is $M = 5$. (a) The s–t path with depicted edge costs. The cost of the path is 1 and its important subpaths are indicated. Relative vertical positions of vertices correspond to costs of subpaths starting or ending at the respective vertex. (b) The SoC function of the s–t path. The coordinates of its breakpoints correspond to the costs of certain important subpaths.

prefix $P_{s,w}$. However, this contradicts the fact that $P_{u,v}$ is the minimum-cost subpath of P, as $P_{w,u} \circ P_{u,v}$ yields a subpath of lower cost. Second, assume that w lies on the subpath $P_{u,v}$. This implies that the u–w subpath $P_{u,w}$ has positive cost, since the prefix $P_{s,u}$ has lower cost than the maximum prefix $P_{s,w}$. Again, this contradicts the fact that $P_{u,v}$ is the minimum-cost subpath of P, since removing its prefix $P_{u,w}$ yields a shorter subpath $P_{w,v}$ from w to v. Third, assume w lies on the subpath $P_{v,t}$. As before, the u–w subpath $P_{u,w}$ must have positive cost in this case, since we would obtain $c(P_{s,w}) < c(P_{s,u})$ otherwise. Since the cost of $P_{u,v}$ is less than $-M$, this means that the cost of the subpath $P_{v,w}$ is greater than M, which contradicts our assumption.

By a symmetric argument, $P_{v,t}$ is the maximum suffix of P. Consequently, if the initial SoC $b_s \in [0, M]$ at the source s is below the cost $c(P_s^+)$ of $P_s^+ = P_{s,u}$, the path is infeasible. Otherwise, the SoC is nonnegative at u. The SoC can only increase when traversing the subpath $P_{u,v}$, since this subpath has no positive prefix (by the assumption that it is the minimum-cost subpath of P). Moreover, the SoC has reached the maximum $b_v = M$ at v, independent of b_s. We also know that the SoC is always below b_v while traversing the v–t subpath $P_{v,t} = P_t^+$, since this subpath has no negative prefix (otherwise, we could use this negative prefix of P_t^+ to find a shorter subpath than $P_{u,v}$, contradicting our assumption that $P_{u,v}$ is the minimum-cost subpath of P). Thus, no constraints apply on this subpath and the SoC at t is $M - c(P_t^+)$, subtracting exactly the (positive) cost of the remaining subpath from v to t.

Case 3: The cost of every subpath of P is in the interval $[-M, M]$. This implies that at any vertex on P, a fully charged battery is sufficient to reach the target, because the SoC cannot drop below 0 after it reached the maximum M. Therefore, depending on the initial SoC $b_s \in [0, M]$, the path may either be infeasible or recuperation is disabled at some point because the maximum SoC is reached, but not both. Based on this observation, we discuss possible SoC values at the target.

First, the path P is infeasible (for some initial SoC $b_s \in [0, M]$) if and only if the SoC value drops below 0 at some vertex v on P. This implies that recuperation is always possible. Thus, no battery constraints apply at any vertex u on the subpath from s to v, so the SoC at each such vertex u is $b_s - c(P_{s,u})$. Consequently, v is the first vertex on P such that this difference becomes negative, i. e., $b_s - c(P_{s,v}) < 0$. Independent of the initial SoC, this difference is minimized at the last vertex of the maximum prefix. It follows that $f_P(b_s) = -\infty$ if and only if $b_s < c(P_s^+)$.

Second, if full recuperation is not possible along some edge (u, v) of P, the path is feasible and the difference between the initial SoC b_s and the cost of the s–v subpath $P_{s,v}$ exceeds the battery capacity, i. e., $b_s - c(P_{s,v}) > M$. For any value $b_s \in [0, M]$, this difference is maximized at the last vertex of the minimum prefix, so the constraint on recuperation applies if and only if $b_s - c(P_s^-) > M$. If this is the case, the SoC reaches the maximum value M at the last vertex of the minimum prefix (after applying battery constraints). Since the remaining maximum suffix P_t^+ has nonnegative cost of at most M and no negative prefix (otherwise, we could use this prefix to extend the minimum prefix of P), it follows that no battery constraints apply on this subpath and the SoC at the target is $M - c(P_t^+)$ if $b_s > M + c(P_s^-)$.

Third, we have argued that if $c(P_s^+) \leq b_s \leq M + c(P_s^-)$, no constraints apply. Therefore, the path is feasible and recuperation is always possible. This implies that the SoC at the target is exactly $b_s - c(P)$. It remains to show that this is the result of evaluating the piecewise linear function defined above. Recall that we have the equality $c(P) = c(P_s^+) + c(P_t^-) = c(P_s^-) + c(P_t^+)$. We obtain that the slope of the function f_P on the interval $[c(P_s^+), M + c(P_s^-)]$ is

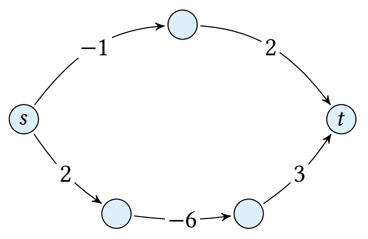
$$\sigma_1 := \frac{y_2 - y_1}{x_2 - x_1} = \frac{M - c(P_t^+) + c(P_t^-)}{M + c(P_s^-) - c(P_s^+)} = 1,$$

where (x_1, y_1) and (x_2, y_2) denote the two breakpoints of the piecewise linear function f_P according to the lemma. Consequently, the function f_P evaluates to

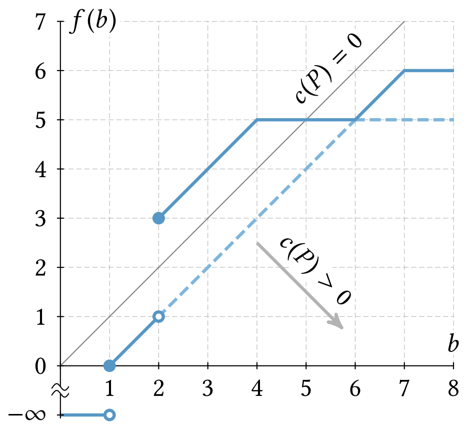
$$f_P(b_s) = y_1 + \sigma_1(b_s - x_1) = -c(P_t^-) + (b_s - c(P_s^+)) = b_s - c(P)$$

for arbitrary $b_s \in [c(P_s^+), M + c(P_s^-)]$, which completes our proof. □

According to Lemma 4.1, the SoC function of a path has a characteristic form: It consists of a first part with infinite consumption (the path is infeasible for low SoC), followed by a segment with slope 1 (the consumption is constant, thus SoC at t increases with SoC at s), and a last segment of constant SoC (for high values of initial SoC, the battery is fully charged at some point due to recuperation). Each of



(a)



(b)

Figure 4.3: The SoC profile of two vertices in a graph. The battery capacity is $M = 8$. (a) The graph with indicated source s and target t. There are two different s–t paths with respective costs 1 and −1. (b) The corresponding SoC function. The dashed segments indicate dominated parts of SoC functions of either of the two s–t paths. Characteristic segments of contributing paths follow the gray arrow in increasing order of their total path length (unless they contain a subpath of cost below $-M$; c. f. Lemma 4.1).

these three parts may collapse to a single point. The segment with slope 1 is also called the *characteristic* segment of the SoC function. An example of a path and its SoC function is depicted in Figure 4.2.

In summary, at most two breakpoints are necessary to represent the SoC function of a path. This stands in contrast to profiles in time-dependent route planning, where profiles map departure time to arrival time in a network with time-dependent edge costs. Such time-dependent profiles can become significantly more complex, even for single paths [Bat+13, Bau+16f, DW09].

Unrestricted Profiles. For a fixed pair of vertices $s \in V$ and $t \in V$, different paths may be the optimal choice for different values of initial SoC; see Figure 4.3 for an example. Consequently, a profile may be composed of multiple paths. A general SoC function is the upper envelope of a set of SoC functions, each corresponding to a single path. Note that this upper envelope may contain multiple discontinuities; see Figure 4.3. Next, we investigate the complexity of such *general* SoC functions. For the sake of simplicity, we assume in the remainder of this section that shortest paths (with respect to the cost function c) between arbitrary pairs of vertices are unique.

We say that an s–t path and its SoC function *contribute* to the s–t profile if they are optimal for some initial SoC. First, we bound the number of breakpoints in the SoC function subject to the number of contributing paths. The following Lemma 4.2 is a direct implication of the observations by Atallah [Ata85]. A more direct proof is given below for the sake of self-containedness. (Note that the number of breakpoints in the upper envelope of linear functions can be superlinear in general [WS88].)

Lemma 4.2. *Given the set $\mathcal{P}$ of all contributing paths of an s–t profile, the number of breakpoints in the corresponding SoC function is linear in $|\mathcal{P}|$.*

Proof. Given the set $\mathcal{P}$ of contributing paths, we construct a sequence F representing the SoC function f of the s–t profile in the following way. Starting with an empty sequence $F = \emptyset$, we iteratively *merge* F with the breakpoints F_P of the SoC function of a contributing path P from $\mathcal{P}$, i. e., we replace F by the upper envelope of the functions defined by F and F_P.

For the sake of simplicity, assume that the SoC function of every contributing path contains exactly two breakpoints. We select paths in $\mathcal{P}$ in decreasing order of their cost with respect to the function c (inverse to the arrow depicted in Figure 4.3b). Thus, when a path $P \in \mathcal{P}$ is chosen, its SoC function f_P dominates the function defined by the current sequence F along its characteristic segment. Hence, the two breakpoints representing this segment are added to F, creating a discontinuity. Moreover, the flat segment following the characteristic segment may intersect at most one segment in F, because F corresponds to an *increasing* function. Such an intersection requires one additional breakpoint in F. In total, incorporating f_P results in a constant number of new breakpoints, which implies that the size of F is linear in the cardinality of $\mathcal{P}$. It is straightforward to generalize the procedure to also handle paths with functions composed of only a single breakpoint. □

Since the number of s–t paths can be exponential in the graph size, Lemma 4.2 does not yield an immediate polynomial bound on the complexity of the s–t profile. We now show that the number of breakpoints in any SoC function is in fact *linear* in the number of vertices of the input graph in the worst case.

Before we prove the bound, we derive basic properties of contributing SoC functions. As argued above, certain subpaths of an s–t path P are relevant to determine its profile. We add the following definitions that are helpful in our further examination. The *bottom vertex* v^- is the last vertex of the minimum prefix (and the first vertex of the maximum suffix) of P. Similarly, the *top vertex* v^+ denotes the last vertex of the maximum prefix (and the first vertex of the minimum suffix) of P. We call v^- and v^+ the *important* vertices of P. We presume that $v^- \neq v^+$, which always holds except in the trivial case $s = t$. The important vertices separate P into three subpaths. (In case that s or t are important vertices, one or two of these subpaths may consist of a single vertex.) Moreover, we distinguish two *types* of s–t paths, depending on the order of appearance of their important vertices. A path P is called *bottom-top path* if v^- appears before v^+ on P, otherwise it is a *top-bottom path.*

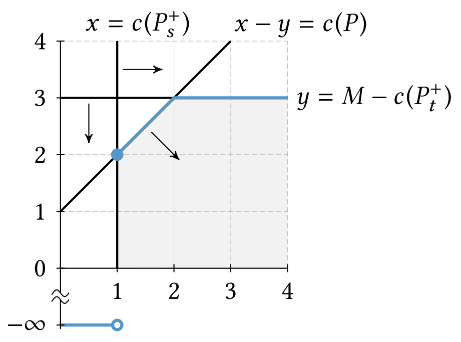


Figure 4.4: Dominated area of an SoC function, for $M = 4$ and a path P with $c(P) = -1$. Its important subpaths have cost $c(P_s^+) = 1$ and $c(P_t^+) = 1$. The costs induce three lines, each of which subdivides the Euclidean plane into two half planes. The SoC function of a path Q with $c(Q) \geq c(P)$, $c(Q_s^+) \geq c(P_s^+)$, and $c(Q_t^+) \geq c(P_t^+)$ lies in the shaded intersection of three of these half planes.

We continue with some basic properties of paths and their SoC functions. First, Lemma 4.3 claims that a path P dominates another path Q if it is shorter (with respect to the cost function c) and both its maximum prefix and its maximum suffix are shorter than the respective subpaths of Q. This follows immediately from the structure of SoC functions according to Lemma 4.1 and is illustrated in Figure 4.4.

Lemma 4.3. *Given two vertices $s \in V$ and $t \in V$, let P and Q be two s–t paths such that $c(P) \leq c(Q)$, $c(P_s^+) \leq c(Q_s^+)$, and $c(P_t^+) \leq c(Q_t^+)$. Then the SoC function f_P of P dominates the SoC function f_Q of Q.*

The next Lemma 4.4 states that prefixes and suffixes of all contributing paths are uniquely defined by their corresponding important vertices.

Lemma 4.4. *Given two vertices $s \in V$ and $t \in V$, let $v \in V$ be an arbitrary fixed vertex. All paths of the same type contributing to the s–t profile with v as their first important vertex share the same s–v subpath. Similarly, all contributing paths of the same type with v as their second important vertex share the same v–t subpath.*

Proof. Assume for contradiction that there are two contributing paths P and Q of the same type, such that the first important vertex of each path is v, but their respective s–v subpaths differ. Without loss of generality, let the s–v subpath of P be shorter. We replace the s–v subpath of Q by the s–v subpath of P, which yields a modified path Q'. Clearly, the length of Q' is below the length of Q, i. e., $c(Q') < c(Q)$. At the same time, neither the maximum prefix nor the maximum suffix of Q' exceeds the cost of the respective subpath of Q. By Lemma 4.3, the modified path Q' dominates Q, contradicting the assumption that Q is a contributing path.

Similarly, we can replace the v–t subpath in one of two paths of the same type that share the second important vertex v by a shorter v–t subpath. Again, we obtain a new path that is shorter, while the lengths of its maximum prefix and suffix do not increase. Hence, at least one of the two paths does not contribute to the profile. □

Using similar arguments, it is straightforward to extend Lemma 4.4 and show that together with their order in the path, pairs of important vertices uniquely define contributing paths of the same type. Note that this already implies that there are at most $O(n^2)$ paths contributing to an s–t profile. We formally prove the claim in Lemma 4.5 below. Afterwards, we use a somewhat more sophisticated argument to show that the number of breakpoints is at most linear in the number of vertices.

Lemma 4.5. *Let $s \in V$, $t \in V$, $v^- \in V$, and $v^+ \in V$ be four vertices of the input graph. There is at most one bottom-top path contributing to the s–t profile that has v^- as its bottom vertex and v^+ as its top vertex. Similarly, at most one contributing top-bottom path has v^+ as its top vertex and v^- as its bottom vertex.*

Proof. Assume for contradiction that there exist two distinct contributing s–t paths P and Q, such that both are bottom-top paths, their bottom vertex is v^-, and their top vertex is v^+. By Lemma 4.4, we know that P and Q share the same s–v^- subpath and the same v^+–t path. Hence, their v^-–v^+ subpaths must differ. Without loss of generality, let the v^-–v^+ subpath of P be shorter. Apparently, the total cost of the path P is lower than the cost of Q, i. e., $c(P) < c(Q)$. Similarly, the cost of the maximum prefix P_s^+ (suffix P_t^+) of P is less than the cost of the maximum prefix Q_s^+ (suffix Q_t^+) of Q. By Lemma 4.3, this implies that P dominates Q, contradicting the fact that Q contributes to the optimal solution. The other case is symmetric, so the claim follows. □

We are now ready to present the main result of this section. Theorem 4.6 proves that the number of breakpoints of an arbitrary SoC function is at most linear in the number of vertices in the input graph. Further, it is easy to construct an example where the SoC function indeed has a linear number of breakpoints; see Figure 4.5. Hence, the bound of Theorem 4.6 is tight up to a constant factor and the number of breakpoints of an SoC function is in $\Theta(n)$ in the worst case.

Theorem 4.6. *Given a source $s \in V$ and a target $t \in V$ in the input graph, the number of contributing paths (and breakpoints) in the s–t profile is in $O(n)$.*

Proof. We construct an undirected graph G' consisting of vertices representing important vertices in the input graph $G = (V, E)$. Every edge of G' represents a contributing path using the corresponding pair of important vertices. We examine the structure of SoC functions of contributing paths to show that the number of edges in the constructed graph is in $O(n)$. Together with Lemma 4.2, this proves our claim.

The graph G' consists of the union of four sets of vertices $V_1^- = \{v_1^- \mid v \in V\}$, $V_1^+ = \{v_1^+ \mid v \in V\}$, $V_2^- = \{v_2^- \mid v \in V\}$, and $V_2^+ = \{v_2^+ \mid v \in V\}$. Clearly, the number of vertices in G' is linear in the number of vertices in the original graph G. We add one undirected edge for every s–t path in the original graph that contributes to the SoC profile: For every contributing bottom-top path with first important vertex $u \in V$ and second important vertex $w \in V$, we add the edge $\{u_1^-, w_2^+\}$. For every contributing top-bottom path with first important vertex $u \in V$ and second important vertex $w \in V$, we add the edge $\{u_1^+, w_2^-\}$. Lemma 4.5 implies that there are no multi-edges in the

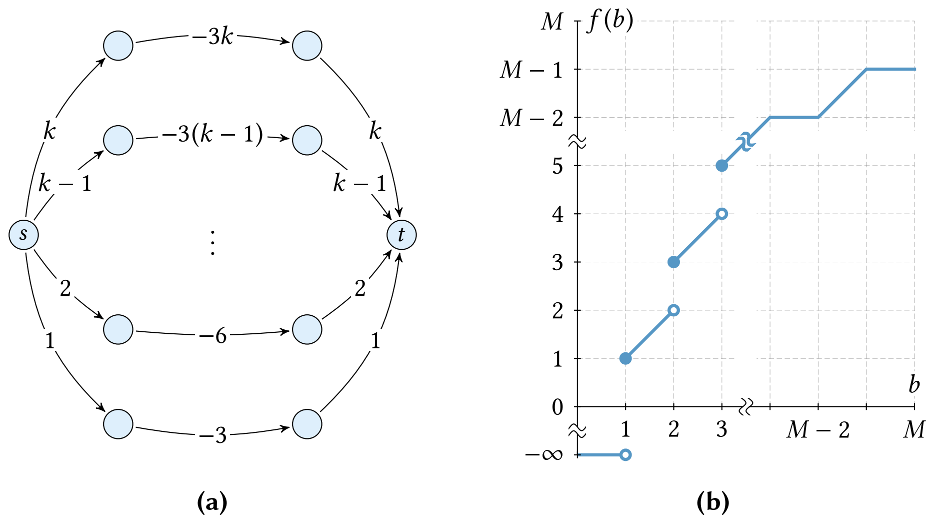


Figure 4.5: An SoC function with $\Theta(n)$ breakpoints. (a) The input graph with designated vertices s and t. There are $k \in \mathbb{N}$ distinct s–t paths and $n = 2(k + 1)$. Edges are labeled with their costs. (b) A sketch of the s–t profile for an arbitrary battery capacity $M \geq 3k$. Every s–t path in the graph contributes to the profile and adds three breakpoints as well as a discontinuity (represented by a fourth breakpoint), which results in an SoC function with $2(n - 3)$ breakpoints in total.

resulting graph. By construction, G' consists of at least two components and each component induces a bipartite subgraph. We claim that G' contains no *simple* cycles, i. e., there is no cycle in G' that uses every edge at most once. This implies that G' has at most $O(n)$ edges, which proves the theorem.

Assume for contradiction that there is a simple cycle $C = [v_1, \ldots, v_k, v_1]$ in the graph constructed above. There are two possible cases: Either all edges in the cycle correspond to top-bottom paths and it contains only vertices in $V_1^+ \cup V_2^-$, or all edges correspond to bottom-top paths and all its vertices are in the set $V_1^- \cup V_2^+$.

Case 1: All edges represent top-bottom paths, and therefore $\{v_1, \ldots, v_k\} \subseteq V_1^+ \cup V_2^-$. Figure 4.6a shows an example. Consider the profile induced by all paths corresponding to the edges of this cycle. Edges incident to some vertex $v_i \in V_1^+$, with $i \in \{1, \ldots, k\}$, correspond to paths with the same top vertex in G. Lemma 4.4 implies that these paths also share the same maximum prefix with some length $x \in [0, M]$. Therefore, by Lemma 4.1, every edge incident to v_i corresponds to some SoC function whose first breakpoint has the x-coordinate x. Thus, the leftmost point of the characteristic segment of each of these SoC functions lies on a vertical line defined by x; see Figure 4.6b. Similarly, edges incident to a bottom vertex $v_i \in V_1^-$ represent paths with the same maximum suffix of length $y \in [0, M]$. The last breakpoint of each SoC function associated with one of these paths lies on a horizontal line defined by the

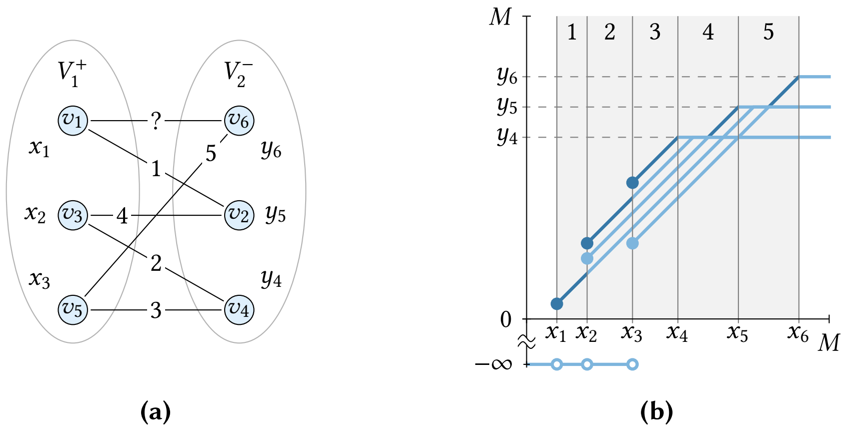


Figure 4.6: Illustration of the proof of Theorem 4.6. (a) A constructed bipartite graph G' with copies of top and bottom vertices of the input graph. Edges represent paths connecting certain important vertices. Vertices have assigned real-valued constants $x_1, x_2, x_3, y_4, y_5, y_6$. Edge labels indicate the interval in Figure 4.6b where the corresponding characteristic segment is contained in the upper envelope of all functions. (b) The SoC functions of the edges in the constructed graph. Characteristic segments connect vertical lines induced by constants $x_1, \ldots, x_6$. Parts of characteristic segments that lie on the upper envelope are highlighted (dark blue). Adding the missing characteristic segment that connects the lines induced by x_1 and x_6 (to form a cycle in the graph) results in at least one dominated SoC function.

y-coordinate y. Hence, each of the k vertices defines either a vertical or a horizontal line. Every edge in the cycle C corresponds to a characteristic segment that starts at one of the vertical lines and ends at one of the horizontal lines; see Figure 4.6b.

For a constant $y \in [0, M]$ inducing a horizontal line, we consider the leftmost x-coordinate of any breakpoint of an SoC function (corresponding to an edge in the cycle C) with the y-coordinate y; see Figure 4.6b. In total, we defined one x-coordinate for each vertex in C, which we denote by $x_i \in [0, M]$ for $i \in \{1, \ldots, k\}$. Without loss of generality, assume that $x_1 < x_2 < \cdots < x_k$ holds. Then, we obtain $k - 1$ intervals $[x_i, x_{i+1}]$ for $i \in \{1, \ldots, k-1\}$. By assumption, every edge of C corresponds to a contributing path. The characteristic segment of the SoC function of each contributing path is (partially) contained in the upper envelope of the SoC functions of all k paths (or else it would not contribute to the s–t profile). Given that all characteristic segments are parallel (with slope 1), this implies that there is a unique segment that is part of the upper envelope on the interval $[x, x_{i+1}]$, with $i \in \{1, \ldots, k-1\}$. However, there are only $k - 1$ such intervals for k contributing paths; a contradiction.

Case 2: All edges represent bottom-top paths, and therefore $\{v_1, \ldots, v_k\} \subseteq V_1^- \cup V_2^+$. In this case, edges incident to a bottom vertex $v_i \in V_1^-$ for some $i \in \{1, \ldots, k\}$ correspond to paths with the same bottom vertex in G. By Lemma 4.4, these paths

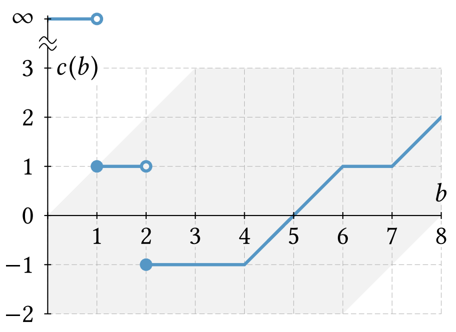


Figure 4.7: A consumption function, corresponding to the s–t profile in Figure 4.3b. The shaded area indicates those values that fulfill the battery constraints.

share the same minimum prefix with length $y \in [0, M]$. Moreover, observe that a contributing bottom-top path contains no subpath with cost below $-M$, since the cost of its maximum prefix must not contain a subpath of length greater than M. It follows that SoC functions of contributing bottom-top paths are of the form as in Case 3 of Lemma 4.1. Thus, the leftmost point of the characteristic segment of each SoC function represented by an edge incident to the bottom vertex v_i lies on the horizontal line defined by y. Similarly, edges incident to top vertices $v_i \in V_1^+$ for some $i \in \{1, \ldots, k\}$ correspond to characteristic segments whose rightmost point lies on the same vertical line defined by a constant $x \in [0, M]$. If we proceed along the lines of the first case, this yields a contradiction. □

Consumption Profiles. An SoC function f maps initial SoC at a source vertex $s \in V$ to the (optimal) resulting SoC at a target vertex $t \in V$. An alternative notion that is common in the literature [Bau+13a, Bau+15b, EFS11] utilizes *consumption functions* $c\colon [0, M] \cup \{-\infty\} \to [-M, M] \cup \{\infty\}$ of pairs of vertices. A consumption function maps initial SoC to minimum *energy consumption*, defined as the difference $b_s - b_t$ between the SoC at s and t, respectively. For arbitrary $b \in [0, M]$, we obtain the relation $c(b) = b - f(b)$ between the consumption function c and the SoC function f with respect to s and t. Figure 4.7 shows the consumption function that corresponds to the SoC function in Figure 4.3b. Apparently, both notions represent different points of view for the same problem. Hence, insights about the structure of SoC functions and bounds on their complexity carry over to consumption functions immediately.

4.1.3 Operations on Profiles

Now that we discussed important properties of SoC functions, we introduce basic operations that enable us to compose and merge SoC functions. These operations are used in Section 4.2.2 to derive algorithmic approaches for the computation of SoC profiles between arbitrary pairs of vertices.

To compute SoC profiles, we require binary *link* (composition) and *merge* operations, defined on the function space $\mathbb{F}$ of SoC functions. Given the SoC functions of two paths P and Q in the input graph, the link operation computes the SoC function of their concatenation $P \circ Q$, i. e., it maps initial SoC to the resulting SoC after traversing P and Q in this order. Formally, the operation $\text{link}\colon \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ takes as input two SoC functions f_1, f_2 and is defined as $\text{link}(f_1, f_2) := f_2 \circ f_1$. Hence, linking f_1 and f_2 yields a new SoC function f, such that $f(b) = f_2(f_1(b))$ for every $b \in [0, M]$. The operation $\text{merge}\colon \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ computes the pointwise maximum of two functions, i. e., merging two SoC functions f_1 and f_2 yields $\text{merge}(f_1, f_2) := \max(f_1, f_2)$. The result is a new SoC function f with $f(b) = \max\{f_1(b), f_2(b)\}$ for every $b \in [0, M]$. Furthermore, our algorithms use *dominance tests* to identify dominated SoC functions during a search. Observe that such a test can be implemented by making use of the merge operation, since an SoC function f_1 dominates another SoC function f_2 if and only if $\text{merge}(f_1, f_2) = f_1$.

The function space $\mathbb{F}$ of SoC functions is closed under the operations link and merge, i. e., the result of each operation is another SoC function. The SoC function f_P of a path $P = [v_1, \dots, v_k]$ is obtained by iteratively applying the link operation, starting with the SoC functions $f_{(v_i, v_{i+1})}$ of its edges (v_i, v_{i+1}) for all $i \in \{1, \dots, k-1\}$. Thus, we get

$$f_P = \text{link}(\dots \text{link}(\dots \text{link}(f_{(v_1,v_2)}, f_{(v_2,v_3)}), \dots), f_{(v_{k-1},v_k)}).$$

Note that linking is not commutative but associative, so the link operations above can be applied in arbitrary order.

In the same fashion, link and merge operations can be derived for consumption functions. Given two consumption functions c_1 and c_2, the link operation is defined as $\text{link}(c_1, c_2) := c_1 + c_2 \circ (\text{id} - c_1)$, whereas we define merging as the pointwise minimum denoted $\text{merge}(c_1, c_2) := \min(c_1, c_2)$. The remainder of this section deals with efficient implementations of these operations. While we focus on SoC functions, the link and merge operation for consumption functions are analogous.

Implementation of Linking and Merging. In general, SoC functions are piecewise linear, but not necessarily continuous, with varying degree of complexity. We propose different representations of SoC functions, depending on their complexity.

For a single edge $e \in E$, the SoC function f_e has a *simple form*: As described in Section 4.1.2, the SoC function f_e is defined only by the constant value $c(e)$. More generally, SoC functions have simple form (i. e., they are defined by a finite constant value as described in Section 4.1.2) if and only if they correspond to an s–t path P with important vertices s and t. In this case, the cost of each important subpath is either 0 or equal to the cost $a := c(P)$ of the whole path. Hence, a single constant $a \in \mathbb{R}$ is sufficient to represent the SoC function. However, linking two functions of simple form does not yield another function of simple form in general: As shown in Section 4.1.2, battery constraints may impose more complex functions when concatenating edges. In fact, linking two functions represented by constants $a_1 \in \mathbb{R}$ and $a_2 \in \mathbb{R}$ yields a function of simple form if and only if both have the same sign, i. e., either $a_1 \geq 0$ and $a_2 \geq 0$ hold or $a_1 \leq 0$ and $a_2 \leq 0$ hold. Then, the resulting SoC function is represented by the constant $a_1 + a_2$, so the link operation boils down to a single addition and a check testing whether the path is always infeasible (if and only if $a_1 + a_2 > M$). Conversely, merging two functions of simple form represented by the respective constants $a_1 \in \mathbb{R}$ and $a_2 \in \mathbb{R}$ always results in a function that has simple form as well, represented by the constant $\min\{a_1, a_2\}$.

As shown in Section 4.1.2, any path P in the graph has an SoC function f_P that can be represented by at most two breakpoints. Note that explicitly storing both breakpoints is redundant, due to the specific form of such profiles (in particular, their slope is always 0 or 1). A slightly more compact representation [EFS11] uses only three values to represent f_P, namely, the minimum SoC $\mathrm{in}_P \in [0, M]$ required to traverse P, its energy consumption $\mathrm{cost}_P \in [-M, M]$ (which can be less than in_P due to recuperation), and the maximum possible SoC after traversing P, denoted $\mathrm{out}_P \in [0, M]$. The value of in_P is the length of the maximum prefix of P, while out_P is the difference between the battery capacity M and the length of the maximum suffix of P. The value cost_P equals the total length of P (unless P contains a subpath with cost below $-M$; see Lemma 4.1 and Figure 4.8). In other words, in_P and cost_P determine the first breakpoint of the SoC function, while out_P is the y-coordinate of its last breakpoint. Thus, the SoC function f_P of the path evaluates to

$$f_P(b) = \begin{cases} -\infty & \text{if } b < \mathrm{in}_P, \\ \mathrm{out}_P & \text{if } b - \mathrm{cost}_P > \mathrm{out}_P, \\ b - \mathrm{cost}_P & \text{otherwise.} \end{cases}$$

For an edge $e \in E$, the SoC function f_e is defined by $\mathrm{cost}_e := c(e)$, $\mathrm{in}_e := \max\{0, c(e)\}$, and $\mathrm{out}_e := \min\{M, M - c(e)\}$. The SoC profile $f_{[v]}$ of a path $[v]$ consisting of a single vertex $v \in V$ is the identity function, which is represented by $\mathrm{cost}_{[v]} := 0$, $\mathrm{in}_{[v]} := 0$, and $\mathrm{out}_{[v]} := M$. For two SoC profiles f_P and f_Q of given paths P and Q, we obtain (in constant time) the linked function $f_{P \circ Q} := \mathrm{link}(f_P, f_Q)$ by setting

$$\begin{aligned} \mathrm{in}_{P \circ Q} &:= \max\{\mathrm{in}_P, \mathrm{cost}_P + \mathrm{in}_Q\}, \\ \mathrm{out}_{P \circ Q} &:= \min\{\mathrm{out}_P - \mathrm{cost}_Q, \mathrm{out}_Q\}, \\ \mathrm{cost}_{P \circ Q} &:= \max\{\mathrm{cost}_P + \mathrm{cost}_Q, \mathrm{in}_P - \mathrm{out}_Q\}, \end{aligned}$$

provided that $\mathrm{out}_P \geq \mathrm{in}_Q$. Otherwise, there exists a subpaths of length greater than the capacity M and the path is infeasible for arbitrary SoC, which implies $f_{P \circ Q} \equiv -\infty$ (this function can be represented by setting, e. g., $\mathrm{cost}_{P \circ Q} := \infty$). An example of two SoC

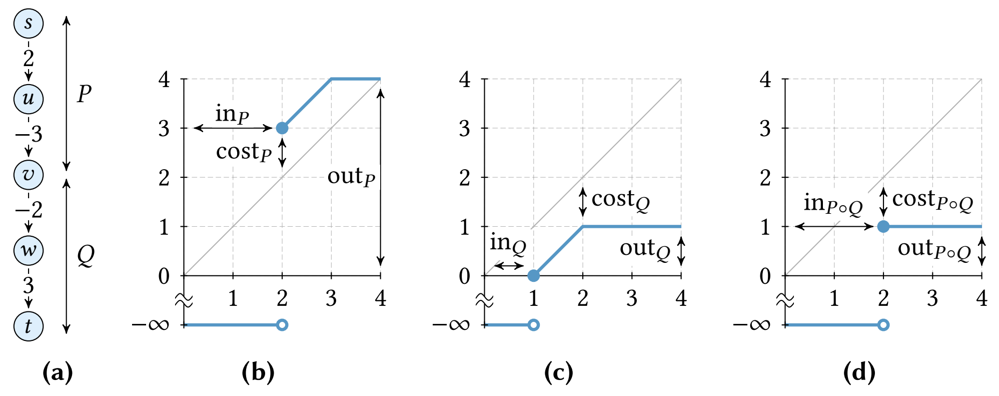


Figure 4.8: Two SoC functions and the result after linking them. The battery capacity is $M = 4$. (a) The underlying paths $P = [s, u, v]$ and $Q = [v, w, t]$, with indicated edge costs. (b) The SoC function f_P is represented by $\text{in}_P = 2$, $\text{cost}_P = -1$, and $\text{out}_P = 4$. (c) The SoC function f_Q is given by $\text{in}_Q = 1$, $\text{cost}_Q = 1$, and $\text{out}_Q = 1$. (d) Linking the functions f_P and f_Q yields the function $f_{P\circ Q}$ with $\text{in}_{P\circ Q} = 2$, $\text{cost}_{P\circ Q} = 1$, and $\text{out}_{P\circ Q} = 1$. Observe that due to a subpath of length $-5 < -M$, the value $\text{cost}_{P\circ Q} = 1$ is greater than the sum $\text{cost}_P + \text{cost}_Q = c(P \circ Q) = 0$.

functions as well as the result of linking them is shown in Figure 4.8. We can also test in constant time whether some SoC function f_P dominates another SoC function f_Q, which is the case if and only if $\text{in}_P \le \text{in}_Q$, $\text{cost}_P \le \text{cost}_Q$, and $\text{out}_P \ge \text{out}_Q$.

We showed in Section 4.1.2 that merging SoC functions of different paths may result in functions with more than two breakpoints. Thus, we need efficient link and merge operations for SoC functions of this general form. Both operations can be implemented as coordinated linear scans, following time-dependent approaches [DW09]. Given two SoC functions f_1 and f_2, the link operation constructs the new function $f := f_2 \circ f_1$ as follows. For each breakpoint $(x, y) \in \mathbb{R}^2$ of f_1, a breakpoint $(x, f_2(y))$ is added to f. For every breakpoint $(x, y) \in \mathbb{R}^2$ of f_2, we test whether x is in the image of f_1. If this is the case, we compute $x' := \min\{b \in [0, M] \mid f_1(b) = x\}$ and add the breakpoint (x', y) to f. Unnecessary breakpoints that do not affect the result of evaluating f (in cases where several collinear breakpoints exist) are removed on-the-fly during the linear scan. Similarly, the merge operation takes two SoC functions f_1 and f_2 and identifies all breakpoints on their upper envelope, i. e., any breakpoint $(x, y) \in \mathbb{R}^2$ of f_1 with $y \ge f_2(x)$ and vice versa. Additional breakpoints are necessary at intersections of both functions, while unnecessary collinear points can be removed.

4.2 Basic Algorithms

In this section, we discuss basic algorithms to answer SoC queries (Section 4.2.1) and profile queries (Section 4.2.2) on a given input graph $G = (V, E)$ with a consumption function $c\colon E \to \mathbb{R}$. First, note that an SoC function f fulfills the *first-in-first-out (FIFO) property*, that is, for arbitrary SoC values $b_1 \in [0, M]$ and $b_2 \in [0, M]$ with $b_1 \leq b_2$, it holds that $f(b_1) \leq f(b_2)$, because SoC functions are increasing. As a result, starting with lower SoC never yields higher SoC at the target [EFS11]. This important property enables label-*setting* algorithms.

4.2.1 SoC Queries

Given a source $s \in V$, a target $t \in V$, and an initial SoC $b_s \in [0, M]$ at s, an SoC query asks for an energy-optimal path, i. e., a path with minimum energy consumption. Our baseline approach for such queries is a known (label-correcting) variant of Dijkstra's algorithm [Art+10a, Art+10b, Joh73], which we refer to as *EV Dijkstra (EVD)*. See Figure 4.9 for pseudocode. Along the lines of Dijkstra's algorithm [Dij59], EVD maintains an *SoC label* $b(v)$ for each vertex $v \in V$, initially set to $-\infty$, except for $b(s)$, which is set to b_s. A priority queue is initialized with the source vertex s and the key $b(s)$. In each step, the main loop scans a vertex $u \in V$ with *maximum* key, extracting it from the queue. Then, for each outgoing edge $e = (u, v) \in E$, the algorithm evaluates the SoC function f_e at SoC $b(u)$. Since SoC functions of edges have simple form, we immediately obtain from Equation 4.1 and Equation 4.2 in Section 4.1.2 that

$$f_e(b(u)) = \begin{cases} -\infty & \text{if } b(u) - c(u, v) < 0, \\ M & \text{if } b(u) - c(u, v) > M, \\ b(u) - c(u, v) & \text{otherwise.} \end{cases}$$

Therefore, scanning an edge requires only a subtraction and two comparisons. If $f_e(b(u)) > b(v)$ holds, the label $b(v)$ is updated accordingly and v is inserted (or updated) in the priority queue. The algorithm stops as soon as the queue runs empty. Then, for each vertex $v \in V$, the label $b(v)$ provably holds the maximum possible SoC b_v when reaching v from s with initial SoC b_s. The minimum energy consumption to reach v equals $b_s - b_v$. (Observe that instead of labels with *maximum* SoC, a more direct adaptation of Dijkstra's algorithm could propagate labels with *minimum* energy consumption.) Correctness follows from the FIFO property [Dre69] and the fact that the above variant of Dijkstra's algorithm correctly handles negative costs [Joh73].

If edges with negative cost exist, the algorithm is label *correcting*: It may scan vertices more than once if their labels are improved via subpaths of negative length. It is well-known that this might trigger (exponentially many) rescans over large parts of the graph [Joh73]. However, this is only the case if negative shortest paths have a long positive prefix (in relation to the graph diameter), which is unlikely in our scenario. Also, recall that our inputs contain no negative cycles due to physical constraints.

Enabling a Stopping Criterion. When solving the SPSP problem for graphs with nonnegative cost functions, the performance of Dijkstra's algorithm can be improved

```
// initialize labels
foreach $v \in V$ do
    $b(v) \longleftarrow -\infty$
$b(s) \longleftarrow b_s$
$Q$.insert($s, b_s$)
// run main loop
while $Q$.isNotEmpty() do
    $u \longleftarrow Q$.deleteMax()
    foreach $(u, v) \in E$ do
        $b \longleftarrow f_{(u,v)}(b(u))$
        if $b > b(v)$ then
            $b(v) \longleftarrow b$
            $Q$.update($v, b$)
```

Figure 4.9: Pseudocode of EVD. The algorithm requires a graph $G = (V, E)$, a cost function $c\colon E \to \mathbb{R}$, a source vertex $s \in V$, a battery capacity $M \in \mathbb{R}_{\geq 0}$, and an initial SoC $b_s \in [0, M]$. For every vertex $v \in V$, it computes the optimal SoC upon arrival at v.

by making use of the stopping criterion; see Section 3.3. For EVD, since it is label correcting, this cannot be applied: After the target vertex t was scanned, there may be vertices (with lower SoC) left in the priority queue. Due to negative costs, it is possible that some of them can be expanded to s–t paths with higher SoC at t. We discuss different ways to establish a stopping criterion for EVD.

Given the target $t \in V$, let P_t^* denote the shortest v–t path in G from any vertex $v \in V$, i. e., the path with cost $c_t^* := c(P_t^*) = \min_{v \in V} \text{dist}_c(v, t)$. We obtain $c_t^* \leq 0$, because the t–t path $[t]$ has cost 0, which yields an upper bound on c_t^*. Assume that, at some point during the execution of EVD, a vertex $v \in V$ is scanned such that $b(v) - c_t^* \leq b(t)$. We claim that at this point, an energy-optimal path was already found, so we can safely abort the search. This is easy to see, as $b(v)$ is the maximum SoC among any labels left in the priority queue. Moreover, $-c_t^*$ is an upper bound on the amount of energy that may possibly be recuperated before reaching t. Hence, the current label $b(t)$ cannot be improved in the further course of the algorithm.

We precompute the value c_t^* for every possible target $t \in V$ to restore the stopping criterion. During an s–t query, this requires an additional check of the condition $b(v) - c_t^* \leq b(t)$ each time a vertex $v \in V$ is scanned. To save space, one can also compute $c^* := \min_{t \in V} c_t^*$ and use only this less accurate bound c^*. Then, instead of n values c_t^* for all $t \in V$, only a single value c^* has to be stored, but the number of vertex scans during queries may increase due to the worse quality of the bound c^*.

It remains to discuss how the bounds c_t^* for all $t \in V$ and c^* are efficiently computed during preprocessing. Assume we (temporarily) add a super source s' and edges (s', v)

with cost 0 for all $v \in V$ to the input graph G. Solving the SSSP problem for s' on this modified graph yields, for each $v \in V$, the length of the shortest s'–v path, which equals the value c^*_v. To compute these paths, we use Dijkstra's algorithm algorithm as described in Section 3.3, which loses its label-setting property in the presence of edges with negative costs [Joh73]. Although it has exponential worst-case running time, it outperforms the polynomial-time algorithm of Bellman-Ford-Moore [Bel58, For56, Moo59] on realistic instances [Art+10a, Art+10b]. If c^* is the desired output, its value can be computed on-the-fly during the search.

The search described above inserts *all* vertices of G into the priority queue in the first iteration of its main loop, since s' is connected to every vertex in the graph. As a result, subsequent queue operations become significantly more expensive. We propose an alternative method that simulates the behavior of the above procedure, but keeps the number of vertices in the queue much smaller. In practice, it is faster by a factor of 2–3. It starts by initializing vertex labels with $d(v) = 0$ for all $v \in V$. This prevents nonnegative labels from being propagated by the search. Then, we process all vertices of the graph sequentially, running Dijkstra's algorithm from each vertex $v \in V$ if $d(v) = 0$ (otherwise, we skip the vertex, because it was already visited by a previous run). We do not reinitialize vertex labels between subsequent runs. Observe that this method behaves exactly like the original algorithm. Hence, it computes the same bounds, but has reduced overhead during queue operations.

Potential Shifting. To get rid of negative costs entirely, we also consider three potential shifting methods, which make EVD label setting. This allows us to apply the regular stopping criterion, i. e., the search is aborted once the target is scanned. Recall that vertex labels in EVD represent the SoC at a vertex. As the algorithm extracts a label with *maximum* key from the priority queue in each step, keys of labels must be *nonincreasing* for the algorithm to become label setting; c. f. Section 3.3.2. Hence, we *subtract* potentials from keys when updating labels and a consistent potential function $\pi \colon V \to \mathbb{R}$ must fulfill the condition $b - \pi(u) \geq f_{(u,v)}(b) - \pi(v)$ for all $(u,v) \in E$ and $b \in [0, M]$. For the SoC function $f_{(u,v)}$ representing an edge $(u,v) \in E$, we know that $f_{(u,v)}(b) \leq b - c(u,v)$ holds for all $b \in [0, M]$ by definition. Therefore, the potential function π should fulfill the condition $c(u,v) - \pi(u) + \pi(v) \geq 0$. In other words, finding a consistent potential for the consumption function c is sufficient to make EVD label setting [EFS11].

The first variant, also proposed by Eisner et al. [EFS11], makes use of a *vertex-induced* potential function $\pi_v \colon V \to \mathbb{R}$. It takes an arbitrary fixed vertex $v \in V$ and sets $\pi_v(u) := \mathrm{dist}_c(u,v)$ for all vertices $u \in V$, i. e., the distance from u to v with respect to the cost function c. Computing these values requires a single run of Dijkstra's (label-correcting) algorithm on the backward graph $\bar{G}$ with cost function c. Consistency of the resulting potential π_v follows immediately from the triangle inequality.

```
// initialize labels
foreach $v \in V$ do
    $f_v \longleftarrow f_{-\infty}$
$f_s \longleftarrow \mathrm{id}$
$Q$.insert($s$,0)
// run main loop
while $Q$.isNotEmpty() do
    $u \longleftarrow Q$.deleteMin()
    foreach $(u,v) \in E$ do
        $f \longleftarrow \mathrm{link}(f_u, f_{(u,v)})$
        if $\exists b \in [0,M]\colon f(b) > f_v(b)$ then
            $f_v \longleftarrow \mathrm{merge}(f_v, f)$
            $Q$.update($v$,key($f_v$))
```

Figure 4.10: Pseudocode of profile search for EVs. The algorithm takes a graph $G = (V,E)$, a cost function $c\colon E \to \mathbb{R}$, a source vertex $s \in V$, and a battery capacity $M \in \mathbb{R}_{\geq 0}$ as input. It computes an s–v profile for every vertex $v \in V$.

The second variant uses of a *bound-induced* potential function $\pi_b\colon V \to \mathbb{R}$, where we simply set $\pi_b(v) := -c_v^*$ for all $v \in V$. The bounds c_v^* are computed by the method described above. Again, consistency follows from the triangle inequality. Cherkassky et al. obtain the same potential function from a multi-source search [Che+10], which preliminary experiments indicated to be slower in our setting.

Finally, we propose a *height-induced* potential $\pi_h\colon V \to \mathbb{R}$. Setting $\pi_h(v) := \alpha \cdot h(v)$ for each vertex $v \in V$, the potential of a vertex depends solely on its elevation $h(v)$ and a constant $\alpha \in \mathbb{R}$. To obtain a consistent potential function, we must determine a value α such that $c(u,v) + \alpha(h(v) - h(u)) \geq 0$ holds for all edges $(u,v) \in E$. If an underlying physical consumption model is known, the value α can often be determined directly from this model [EFS11, Sac+11]. We propose a more general method to compute α, which also works if the underlying model is unknown because edge costs stem from, e. g., simulations or real-world measurements. Given an edge $e = (u,v) \in E$, we denote by $\Delta(e) := h(v) - h(u)$ its *ascent*. Moreover, we define $\alpha_e := -c(e)/\Delta(e)$. It follows that $\alpha \geq \alpha_e$ must hold for all *uphill* edges, i. e., edges with $h(u) < h(v)$. The uphill edge $e \in E$ maximizing α_e yields a lower bound $\underline{\alpha} \in \mathbb{R}$ on the value of α. For *downhill* edges with $h(u) > h(v)$, i. e., edges with negative ascent, we get the condition $\alpha \leq \alpha_e$. This induces an upper bound $\bar{\alpha} \in \mathbb{R}$ on α. If $c(u,v) \geq 0$ holds for all edges $(u,v) \in E$ with $h(u) = h(v)$ and we also obtain $\underline{\alpha} \leq \bar{\alpha}$, an arbitrary value $\alpha \in [\underline{\alpha}, \bar{\alpha}]$ yields a consistent potential π_h. Note that the value α is *negative* for realistic consumption models. Computing $\underline{\alpha}$ and $\bar{\alpha}$ requires only a single linear scan over all edges of the graph, which is also straightforward to parallelize. Moreover, to enable fast integration of new consumption functions in practice, one can improve cache friendliness by precomputing an array that explicitly stores for every edge $e \in E$ the (metric-independent) difference $\Delta(e)$.

It is easy to construct examples where $\underset{\sim}{\alpha} > \bar{\alpha}$ holds, even in the absence of negative cycles. Then, there exists no value $\alpha \in \mathbb{R}$ that allows for a consistent potential function π_h. However, we argue that a consistent potential is found for *realistic* consumption models. Assuming that the velocity is constant along an edge $e \in E$, its energy consumption $c(e)$ has the form $c(e) = \ell \cdot (\lambda_1 + s\lambda_2)$ in common physical models [Asa+16, FAR16, Lv+16, Sac+11], where $\lambda_1 \in \mathbb{R}_{\geq 0}$ and $\lambda_2 \in \mathbb{R}_{\geq 0}$ are *nonnegative* coefficients, and $\ell \in \mathbb{R}_{\geq 0}$ and $s \in \mathbb{R}$ denote the *length* and the *slope* of the road segment represented by e, respectively. Note that $s = \Delta(e)/\ell$. Then, inexistence of a consistent potential π_h implies that there is an *uphill* edge e^+ with coefficients λ_1^+, λ_2^+, length ℓ^+, and *positive* slope s^+ as well as a *downhill* edge e^- with coefficients λ_1^-, λ_2^-, length ℓ^-, and *negative* slope s^-, such that

$$\begin{aligned} & \underset{\sim}{\alpha} \geq \alpha_{e^+} = -\frac{c(e^+)}{\Delta(e^+)} > -\frac{c(e^-)}{\Delta(e^-)} = \alpha_{e^-} \geq \bar{\alpha} \\ \Leftrightarrow \quad & \frac{\ell^+\lambda_1^+}{\Delta(e^+)} + \frac{s^+\ell^+\lambda_2^+}{\Delta(e^+)} < \frac{\ell^-\lambda_1^-}{\Delta(e^-)} + \frac{s^-\ell^-\lambda_2^-}{\Delta(e^-)} \\ \Leftrightarrow \quad & \lambda_2^+ < \frac{\lambda_1^+}{s^+} + \lambda_2^+ < \frac{\lambda_1^-}{s^-} + \lambda_2^- < \lambda_2^-. \end{aligned}$$

In other words, the coefficient λ_2^- that determines energy gained on a downhill ride is greater than the coefficient λ_2^+ that determines loss on an uphill ride. Certainly, such model parameters are not meaningful, since recuperation efficiency is bounded in reality. In particular, physical constraints prevent the amount of recoverable energy on a downhill ride from outweighing the cost when going uphill on the same slope. Consequently, we were always able to compute the potential function π_h from realistic consumption data in our experiments.

4.2.2 Profile Queries

Under some circumstances, e. g., when charging overnight, it is important to know how much charging is at least required to reach the target. As we have seen in Section 4.1.2, charging more than that might even enable paths with lower energy consumption (such as paths of lower overall cost that first go uphill). As charging requires substantial time, such decisions should be made by the driver. Therefore, we discuss *profile search* to compute optimal paths for *every* possible initial SoC. Profile search is also an important ingredient of the speedup techniques we introduce in the next sections.

Although computationally more expensive, profile search is conceptually easy, as we can adapt the label-correcting search described in Section 3.3. For pseudocode of profile search for EVs, see Figure 4.10. Starting from the source vertex $s \in V$, the algorithm maintains, for each vertex $v \in V$, a label f_v that represents an s–v profile taking the general form of an SoC function; recall Figure 4.3 from Section 4.1.2. The algorithm initializes $f_v \equiv -\infty$ for all $v \in V$ except s, for which the SoC function $f_s = \mathrm{id}$ is the identity function (which corresponds to a consumption of 0 for arbitrary SoC). The source s is inserted into the priority queue with its key, defined for an SoC function f as the value $\min_{b \in [0,M]} b - f(b)$, i. e., its *minimum consumption* (thereby, following the order depicted in Figure 4.3b). In each step of the main loop, the algorithm scans a vertex $u \in V$ with minimum key and follows the basic search outlined in Section 3.3. Incident outgoing edges $e = (u, v) \in E$ are scanned by computing $f = \mathrm{link}(f_u, f_e)$ and possibly updating $f_v = \mathrm{merge}(f_v, f)$, using the operations introduced in Section 4.1.3.

Target Pruning. As in EVD, negative costs prevent us from using a stopping criterion in the profile search, since labels may be improved via subpaths of negative length. We can use exactly the same techniques as described in the previous Section 4.2.1 to remedy this issue. For a given target $t \in V$, a naïve adaptation of the stopping criterion proposed in Section 3.3 then checks whether $f_v^{\max} \leq f_t^{\min}$ holds when scanning some vertex $v \in V$. However, we typically obtain $f_t^{\min} = -\infty$, which renders this stopping criterion useless in most cases. Instead, we apply the following *target pruning* rule, which is used in combination with vertex potentials or bounds induced by values c_v^* for all $v \in V$, as introduced in Section 4.2.1. Given an SoC function f_v in the label of some vertex $v \in V$, let $b_v^{\min} \in [0, M] \cup \{\infty\}$ denote the smallest SoC value for which $f_v(b_v^{\min})$ is finite, i. e., $f_v(b) = -\infty$ for some $b \in [0, M]$ if and only if $b < b_v^{\min}$. If no such real value exists (i. e., $f_v \equiv -\infty$), we define $b_v^{\min} := \infty$. Moreover, let $c_v^{\max \leq M} \in [0, M] \cup \{\infty\}$ denote the maximum *finite* consumption of f_v, i. e., the real value that maximizes the energy consumption $c_v(b) = b - f_v(b)$ for $b \in [0, M]$ (we define $c_v^{\max \leq M} := \infty$ if $f_v \equiv -\infty$). Then, whenever the algorithm scans a vertex $v \in V$, it checks whether both $b_v^{\min} \geq b_t^{\min}$ and $c_v^{\min} \geq c_t^{\max \leq M}$ hold. If that is the case, the algorithm *prunes* the search at v, i. e., it does not scan any of its outgoing edges.

Complexity. Even if we use potential shifting, profile search remains label *correcting*. However, nonnegative reduced costs together with a slightly modified key function (for the priority queue) enable us to establish a polynomial bound on its running time. Recall that in the basic profile search described above, the key of a vertex is defined as the minimum consumption of its current SoC profile. Instead, we now construct a key function with the important property that the minimum key in the priority queue cannot decrease during the search. To this end, we assume that a consistent potential function $\pi \colon V \to \mathbb{R}$ is given. We set the key of a vertex $v \in V$ to its potential $\pi(v)$ plus the minimum consumption $x - y$ among all breakpoints $(x, y) \in \mathbb{R}^2$ that were *newly* added to the function f_v during some merge operation since v was scanned for the last time (or all breakpoints of f_v if v has not been scanned so far). This implies that the minimum key in the priority queue cannot decrease during the search, because scanning an edge $(u, v) \in E$ can only lead to new breakpoints at v whose (reduced) consumption value is at least as large as the (reduced) consumption of a corresponding breakpoint at u that was not propagated to v yet. As a result, each time v is extracted from the queue, its SoC function contains some new breakpoint that has the smallest key among any breakpoints that are yet to be propagated by the search. Hence, this breakpoint must be part of the s–v profile, as it cannot be dominated by any label in the queue. Therefore, the number of times a vertex $v \in V$ can be scanned is bounded by the number of contributing paths in the s–v profile. This means that the number of steps in the main loop is bounded by $O(n^2)$. Moreover, the modified key of a vertex is easily determined during the merge operation, by keeping track of the minimum consumption of all breakpoints that are newly added to a label. Thus, we immediately get the following Theorem 4.7.

Theorem 4.7. *Given a source $s \in V$ and a target $t \in V$ in the input graph, profile search computes the s–t profile in polynomial time.*

Implementation Details. In order to achieve best performance in practice, an efficient implementation of piecewise linear functions is crucial. Because of their specific form (discontinuous; slope is always 0 or 1), we implement SoC functions by storing a pair of point and slope for each breakpoint. Since, in practice, an SoC function f consists of few breakpoints on average, it is best to evaluate $f(b)$ for some $b \in [0, M]$ by a linear scan over its breakpoints rather than using more sophisticated methods like binary search.

As an optimization we use a *compressed function representation.* It stores a single 32-bit integer for functions that have simple form (explicitly checking for battery constraints in the algorithm). We implement link and merge by linear scans in general, as described in Section 4.1.3. However, for compressed functions these operations are much simpler. Merging reduces to a (scalar) minimum operation. Linking becomes simpler as well, as it requires only scalar additions and checks for border cases; see Section 4.1.3. We also provide specialized implementations for the cases where exactly one of the two functions has simple form. For example, linking essentially reduces to shifting an SoC function by a constant in this case. Using compressed functions saves about a factor of 2 in running time of profile search.

To improve spatial locality of the profile search, we store vertex labels as a dynamic adjacency array. For each label representing a vertex $v \in V$, it uses a flag to indicate if f_v is a compressed function, storing the (compressed) value directly at the label. Otherwise, it stores (bit-compressed) indices to a *breakpoint array.* Note that the number of breakpoints of f_v may vary during the algorithm. We mark empty slots in the array in order to make efficient (re)use of space.

4.3 Energy-Optimal Routes with Charging Stops

In the previous section, we discussed algorithms for finding energy-optimal routes that take battery constraints into account. However, as battery capacities of EVs are typically rather small, *recharging* can be inevitable on long-distance routes. With the advent of more powerful charging stations, charging stops are also becoming increasingly appealing to customers. Therefore, we now consider the possibility of recharging the battery at designated charging stations.

In what follows, we formally describe how we extend our model and define the problem (Section 4.3.1), before we introduce a basic label-setting algorithm to solve it (Section 4.3.2). We discuss a conceptually simple polynomial-time approach (Section 4.3.3) and propose a framework to implement it efficiently (Section 4.3.4).

4.3.1 Model and Problem Statement

In addition to our previous setting (see Section 4.1.1), we consider stops at charging stations to recharge the battery. In our model, a subset $S \subseteq V$ of the vertices represents charging stations, where the battery can be charged. To model realistic restrictions, every charging station $v \in S$ has a predefined *SoC range* $R_v = [b_v^{\min}, b_v^{\max}] \subseteq [0, M]$ of possible final SoC. In other words, when charging at v with *arrival SoC* $b \in [0, M]$, we have to pick a desired *departure SoC* $b' \in [b_v^{\min}, b_v^{\max}] \cup \{b\}$. It is always allowed to pick the arrival SoC b as departure SoC, to account for the possibility of not charging at v. Otherwise, we only allow the SoC to *increase* when charging, i. e., we assume $b < b'$. Note that this is not a restriction, because due to the FIFO property, voluntarily decreasing the SoC during a ride never pays off [EFS11]. By making use of SoC ranges at charging stations, we are able to model restrictions caused by technical features of charging stations or user preferences. For example, at regular charging stations, users might wish to recharge only up to a certain percentage of the battery capacity to save time (typically, charging becomes more time consuming when the SoC is near the maximum). At swapping stations, on the other hand, we obtain $R_v = [M, M]$.

Assume we are given a source $s \in V$, a target $t \in V$, and the initial SoC $b_s \in [0, M]$. Observe that simply maximizing the SoC at the target is not meaningful in our new setting: To obtain an optimal solution, we would essentially need to search for a charging station that is as close to the target as possible. Instead, we consider the problem of finding a feasible route (respecting battery constraints) that minimizes *overall* energy consumption, defined as the difference $b_s - b_t$ between SoC at s and t, plus the total amount $r_t \in \mathbb{R}_{\geq 0}$ of energy recharged at charging stations $v \in S$ in order to reach t. Hence, our objective is to *maximize* $b_t - r_t$ among all feasible solutions.

There is no straightforward way to generalize the algorithm described in Section 4.2.1 to this setting, for several reasons. First, it may be wasteful to fully recharge the battery at a charging station, since this may prevent recuperation of energy on subsequent

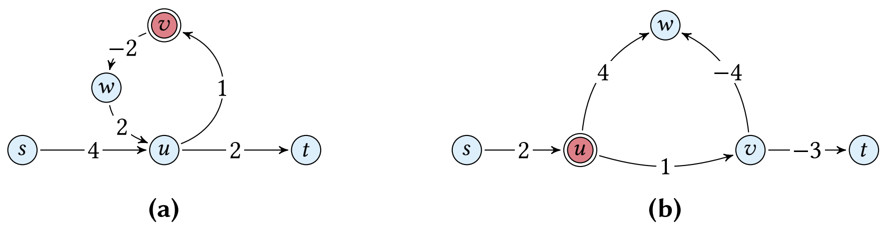


Figure 4.11: Energy-optimal paths with charging stops. The battery capacity is $M = b_s = 5$. Charging stations are highlighted (red) and their SoC range is $[0,5]$. (a) The energy-optimal s–t path $[s,u,v,w,u,t]$ contains a cycle, because a detour to the charging station v is necessary. Recharging any amount $r_t \in [2,3]$ at v yields an SoC $b_t \in [0,1]$ at t, which corresponds to an optimal consumption of 7 and the objective $b_t - r_t = 2$. In all other cases, either energy is wasted or t is not reachable. (b) The optimal s–t path is $[s,u,v,t]$ and t is reached with a full battery and energy consumption 0 without recharging, i. e., $b_t = 5$ and $r_t = 0$. The consumption along the subpath $[s,u,v]$ is 3 and we get $b_v - r_v = 2$. The optimal s–v path $[s,u,w,v]$ requires recharging of at least one unit at u in order to reach v with optimal consumption 2 and objective $b_v - r_v = 3$, where $b_v \in [4,5]$ and $r_v \in [1,2]$.

road segments. As a result, overall consumption may increase for a full battery; see Figure 4.11a for an example. Therefore, we do not know the optimal amount of energy to be charged when a station is scanned. This makes our problem setting significantly more difficult compared to simpler models, which assume that charging always results in a full battery [SF12]. Second, in general, an optimal s–t path does not have the important property that every subpath is an optimal path as well. For example, detours may be necessary to visit a charging station; see Figure 4.11a. One can even construct cases where an optimal path that requires no charging contains a subpath that can be improved via charging; see Figure 4.11b. Below, we propose algorithmic solutions to deal with these challenges. In particular, we show that despite the issues outlined above, the problem is solvable in polynomial time.

4.3.2 Baseline Approach

Apparently, making "greedy" choices locally during the search for an optimal path can lead to suboptimal results. A natural way to deal with this issue is the use of label *sets* that model different situations at vertices, similar to multicriteria scenarios [Han80, Mar84]. We now describe how the multicriteria shortest path algorithm described in Section 3.3.1 can be adapted to our problem setting.

For a query from a source $s \in V$ to a target $t \in V$ with initial SoC $b_s \in [0,M]$, a feasible solution is characterized by the corresponding SoC $b_t \in [0,M]$ at t and the total amount $r_t \in \mathbb{R}_{\geq 0}$ of energy recharged along the way. Recall that the objective is to maximize $b_t - r_t$ among all feasible solutions. To reflect this, vertices maintain labels that store both the current SoC and the amount of recharged energy. However, since charging stations may offer *continuous* SoC ranges, pairs of SoC and the corresponding amount of recharged energy are not sufficient to represent all possible solutions in general. Therefore, a label ℓ stores an *SoC range* $[b_\ell^{\min}, b_\ell^{\max}]$ and a *charging range* $[r_\ell^{\min}, r_\ell^{\max}]$ to reflect different possible choices of recharging at (previous) charging stations and the resulting SoC at the current vertex. As in the multicriteria scenario, we can apply Pareto dominance to remove suboptimal labels; see Figure 4.12a. Observe that explicitly storing all four values $b_\ell^{\min}$, $b_\ell^{\max}$, $r_\ell^{\min}$, and $r_\ell^{\max}$ is redundant; it actually suffices to only keep three of them in the label to determine the last one, similar to SoC functions of paths (c. f. Section 4.1.3).

Using the modified vertex labels, we outline an adaptation of the multicriteria shortest path algorithm. The algorithm is initialized with a source label containing the SoC range $[b_s, b_s]$ and the charging range $[0, 0]$. The label is also inserted into a priority queue. In each step of the main loop, the algorithm extracts a label ℓ with SoC range $[b_\ell^{\min}, b_\ell^{\max}]$ and charging range $[r_\ell^{\min}, r_\ell^{\max}]$ at some vertex $u \in V$ with maximum key from the priority queue (defined for the label ℓ as, e. g., the difference $b_\ell^{\max} - r_\ell^{\max}$). It then checks whether u is a charging station. If this is the case, it merges the SoC range $[b_u^{\min}, b_u^{\max}]$ into ℓ, which yields the range

$$[b_\ell^{\min}, b_\ell^{\max}] \cup [\max\{b_\ell^{\min}, b_u^{\min}\}, \max\{b_\ell^{\min}, b_u^{\max}\}].$$

Similarly, the charging range is extended by the additional amount of energy that can be recharged. Formally, we get the new range

$$[r_\ell^{\min}, r_\ell^{\max}] \cup [r_\ell^{\min} + \max\{b_u^{\min} - b_\ell^{\min}, 0\}, r_\ell^{\min} + \max\{b_u^{\max} - b_\ell^{\min}, 0\}].$$

Note that this may create discontinuities in both ranges; see Figure 4.12b. We resolve this issue by generating a new label at u in such cases, so all labels still have constant complexity. The new label is added to the priority queue, unless it is dominated by some existing label at u. Afterwards, the outgoing edges of u are scanned (regardless of whether u is a charging station). Given the energy consumption $c(u,v)$ of an edge $(u,v) \in E$, we know that it cannot be traversed if $b_u^{\max} - c(u,v) < 0$, so no new label is generated in this case. Otherwise, we apply battery constraints and obtain the new range

$$[\max\{b_u^{\min} - c(u,v), 0\}, \min\{b_u^{\max} - c(u,v), M\}]$$

for the new label generated at v. Similarly, battery constraints may affect the charging range, so we shrink it by dropping values for which the path becomes infeasible or recuperation is hindered; see Figure 4.12c. We obtain a new label, which is added to the label set at v and the priority queue if no label at v dominates it.

After termination of the algorithm, the label set at t contains a label with a pair of SoC b_t and charged energy r_t that maximizes the objective (unless no feasible solution

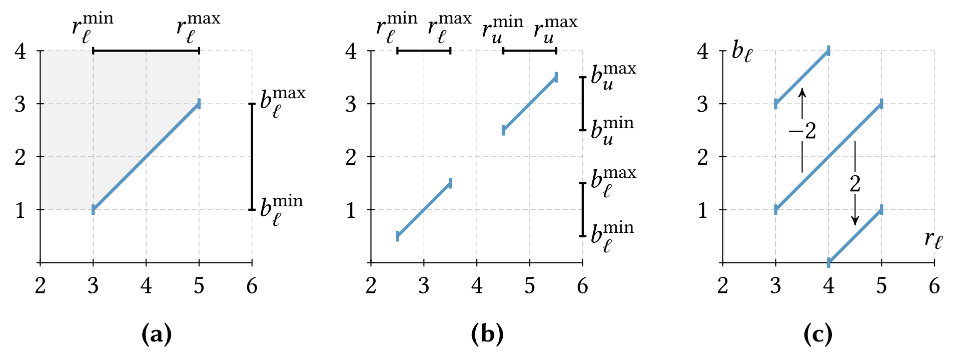


Figure 4.12: Illustration of labels in the baseline approach. They map the amount of charged energy to the resulting SoC, assuming $M = 4$ (charging ranges can exceed the value 4 if multiple charging stops are required). (a) The blue segment shows different configurations of charging and resulting SoC of a label ℓ. Labels are dominated by ℓ if and only if their corresponding segment is entirely contained in the shaded area. (b) Scanning a charging station $u \in S$ adds a new segment with SoC range $[b_u^{\min}, b_u^{\max}]$ to the label, creating a discontinuity. Both segments are collinear, though. (c) Scanning edges (with costs indicated by the arrows) corresponds to shifting the segment along the y-axis. Ranges shrink due to battery constraints, because certain subranges have insufficient charge or waste energy from recuperation.

exists, in which case the label set at t is empty). Correctness follows from the fact that our search propagates all feasible solutions that are not dominated by others. However, due to the complex nature of the algorithm, the analysis of its running time is rather involved [Sau15]. Given that it is based on an exponential-time algorithm, it is not even clear whether its running time is polynomial. In the next section, we present an alternative approach that, building upon tools from Section 4.1, is conceptually simpler, can easily be integrated with known speedup techniques, and runs in polynomial time.

4.3.3 A Polynomial-Time Algorithm

The basic approach described in the previous section uses label sets to model different choices at charging stations. Instead, we now try to *immediately* determine the departure SoC at a charging station, so we only have to maintain a *single* label per vertex. To this end, we first analyze relevant properties of charging stations. Afterwards, we derive an algorithm that maintains one label per vertex on an extended search graph and show that it runs in polynomial time.

Optimal Paths between Charging Stations. At a charging station $u \in S$, the amount of energy that needs to be recharged depends on the route from u to the target $t \in V$, which is not known in advance. Nevertheless, when charging at u, we have to ensure that the SoC is sufficient to reach t or the next charging station $v \in S$.

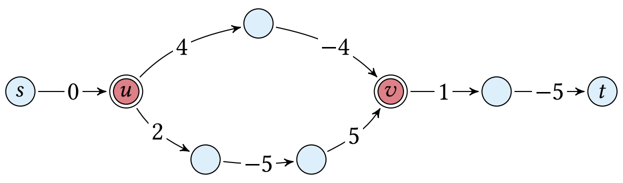


Figure 4.13: Interdependence between initial SoC and the optimal route with charging stops. Charging stations are highlighted (red) and have a charging range of $[0, 5]$, assuming a battery capacity of $M = 5$. Independent of the initial SoC $b_s \in [0, M]$ at the source s, the objective at v is maximized when traversing the u–v path with cost 0. For $b_s \in [0, 4)$, this requires recharging at u (departure SoC $b_u^{\text{dep}} = 4$) and yields $b_v - r_v = b_s$. The target t is always reached with an SoC of $b_t = 5$, so the objective is equivalent to minimizing the amount r_t of charged energy. The optimal choice depends on the value b_s: For $b_s \in [0, 2)$, energy is recharged at u ($b_u^{\text{dep}} = 2$) and v ($b_v^{\text{dep}} = 1$), which yields a total amount $r_t = 3 - b_s > 1$ of recharged energy; for $b_s \in [2, 3]$ it is optimal to charge only at v ($b_v^{\text{dep}} = 1$) to get $r_t = 1$; for $b_s \in [3, 4)$ energy is only charged at u ($b_u^{\text{dep}} = 4$) to get $r_t = 4 - b_s \leq 1$; no charging is necessary at all for $b_s \in [4, 5]$.

Therefore, we examine an important subproblem, where we are given a charging station $u \in S$, an (optimal) arrival SoC $b_u^{\text{arr}} \in [0, M)$ before charging at u, the total amount $r_u \in \mathbb{R}_{\geq 0}$ of energy recharged so far (at any *previous* charging stations), and a vertex $v \in S \cup \{t\}$. We want to find a departure SoC $b_u^{\text{dep}} > b_u^{\text{arr}}$ after charging at u that maximizes the objective at the target vertex t under the assumption that v is the next vertex where energy is recharged or $v = t$ is the target itself. If we compute the u–v profile $f_{u,v}$, we can greedily optimize the objective on the s–v path by picking an SoC $b_u^{\text{dep}} > b_u^{\text{arr}}$ that maximizes $f_{u,v}(b_u^{\text{dep}}) - (r_u + r)$, where $r := b_u^{\text{dep}} - b_u^{\text{arr}}$ is the amount of energy charged at u. Unfortunately, the s–v path that maximizes this objective does not extend to the best solution at t in general. The reason for this is that charging too much energy might prevent the vehicle from recuperating energy on the following v–t path. Figure 4.13 shows an example. Assuming a low initial SoC, the objective at v is maximized in this example when charging to a departure SoC of 4 at the station u. Note that this enables the use of the path with total consumption 0. However, it also prevents recuperation of a significant amount of energy on the subsequent v–t path. Therefore, the objective at t is maximized after charging only to a departure SoC of 2 at u and taking the more expensive subpath from u to v instead.

Apparently, we need a more sophisticated approach. To this end, we identify departure SoC values that may possibly lead to an optimal solution. We know by the FIFO property [EFS11] that for an arbitrary fixed departure SoC $b_u^{\text{dep}} \in [0, M]$, a u–v subpath with minimum energy consumption must be an optimal choice (it cannot be beneficial to pick a more expensive path in order to reach v with a lower SoC). By Theorem 4.6, there are at most $O(n)$ such u–v paths for all possible values of departure SoC, namely, those that contribute to the u–v profile $f_{u,v}$. Moreover, we claim that for each u–v path P contributing to $f_{u,v}$, we can identify a (unique) *canonical*

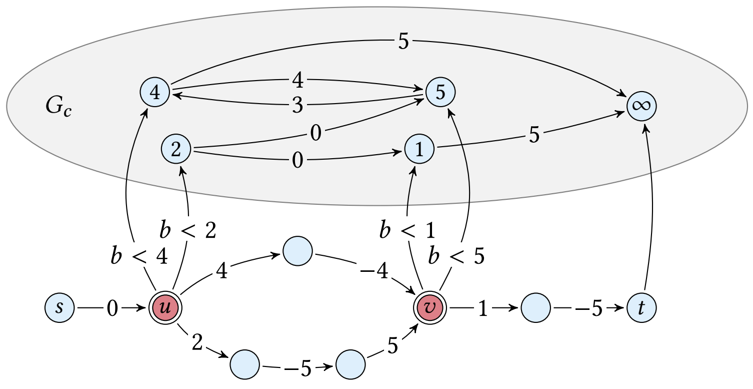


Figure 4.14: Search graph for energy-optimal routes with charging stops, based on the original graph depicted in Figure 4.13. Vertices in the charging station graph (shaded area) are labeled with their departure SoC. Edge labels indicate costs in the original graph, arrival SoC in the charging station graph, and SoC restrictions for transfer edges.

departure SoC $b_P^{\text{dep}} \in [0, M]$ at u that always optimizes the objective at t under the assumption that recharging is necessary at v (or $v = t$). To see this, consider the SoC function f_P of P and let $b_P^{\min} := c(P_u^+)$ denote the minimum SoC that is necessary to traverse P. In other words, $f_P(b) = -\infty$ if and only if $b < b_P^{\min}$. Consequently, we have $b_P^{\text{dep}} \geq b_P^{\min}$. We also know that the objective $f_P(b_P^{\text{dep}}) - (r_u + b_P^{\text{dep}} - b_u^{\text{arr}})$ of the s–v path can only *decrease* for $b_P^{\text{dep}} > b_P^{\min}$, since $r_u - b_u^{\text{arr}}$ is constant and the slope of f_P is at most 1 on the interval $[b_P^{\min}, M]$. Assuming that we are recharging energy at v anyway, charging more than $b_P^{\min}$ will also never turn out to be essential after visiting v: If necessary, we can simply recharge the missing energy at v. Therefore, given the SoC range $[b_u^{\min}, b_u^{\max}]$ of u, we pick the canonical departure SoC $b_P^{\text{dep}} := \max\{b_P^{\min}, b_u^{\min}\}$ for P, if this value lies in the SoC range of u. Otherwise, we have $b_u^{\max} < b_P^{\min}$, which implies that charging at u never renders the path P feasible.

In conclusion, although we cannot compute the optimal u–v subpath without further knowledge about the subsequent v–t path, there is a limited number of candidate paths and for each, there is a unique canonical departure SoC when leaving the charging station u. Moreover, observe that once we fix a departure SoC b_u^{dep} at u, the objectives of the subpaths from s to u and from u to t are *independent* of each other: We obtain an optimal solution via u with departure SoC b_u^{dep} by concatenating an s–u path with maximum objective (subject to the constraint $b_u^{\text{arr}} < b_u^{\text{dep}}$) and a u–t path that maximizes the objective for an initial SoC b_u^{dep}. Based on these observations, we construct a search graph that serves as input for a modified version of EVD.

Search Graph Construction. Given the original graph $G = (V, E)$ and the target vertex $t \in V$, we augment G with a *charging station (sub)graph* $G_c = (V_c, E_c)$, which enables efficient search between charging stations; see Figure 4.14 for an example. The basic idea is to create copies of charging stations for every canonical departure SoC and insert edges that connect feasible sequences of charging stops. For each vertex $u \in S$, we create one *charging vertex* u' per distinct canonical departure SoC $b_P^{\mathrm{dep}} \in [0, M]$ of *any* contributing path P from u to another charging station or to the target. The vertex u' itself is added to V_c. We explicitly store the corresponding departure SoC b_P^{dep} with the vertex u', i. e., we keep a mapping $b^{\mathrm{dep}}\colon V_c \to [0, M] \cup \{\infty\}$ and set $b^{\mathrm{dep}}(u') := b_P^{\mathrm{dep}}$. We also add a dummy target t' to V_c with $b^{\mathrm{dep}}(t') := \infty$.

Edges in the charging station graph represent energy-optimal paths between charging stations. Let P be a (contributing) path from a charging station $u \in S$ to another vertex $v \in S \cup \{t\}$ and f_P its SoC function. We add edges (u', v') from the (unique) vertex $u' \in V_c$ with $b^{\mathrm{dep}}(u') = b_P^{\mathrm{dep}}$ to every charging vertex $v' \in V_c$ of v with $f_P(b_P^{\mathrm{dep}}) < b^{\mathrm{dep}}(v')$ to E_c. At the edge (u', v'), we also store the SoC upon arrival at v', i. e., we store a mapping $b^{\mathrm{arr}}\colon E_c \to [0, M]$ and set $b^{\mathrm{arr}}(u', v') := f_P(b_P^{\mathrm{dep}})$.

The search is run on the union of the input graph G and the charging station graph G_c. To connect both graphs, we add (directed) *transfer edges* (v, v') from each charging station $v \in S \cup \{t\}$ to all its corresponding departure vertices $v' \in V_c$. Transfer edges have no cost, but may only be traversed if the current SoC is below the departure SoC $b^{\mathrm{dep}}(v')$ of the respective departure vertex v', i. e., energy must be recharged to reach the next charging station (or the target). We can model this constraint implicitly, by assigning the SoC function $f_{(v,v')}$ defined as

$$f_{(v,v')}(b) := \begin{cases} b & \text{if } b < b^{\mathrm{dep}}(v'), \\ -\infty & \text{otherwise,} \end{cases} \tag{4.3}$$

to the edge (v, v'). Although this function does not fulfill the FIFO property (as it is not increasing), correctness is maintained because charging edges only control transfer to the charging station graph. A path with departure SoC $b^{\mathrm{dep}}(v')$ may still be traversed in the original graph (without recharging at v) if $b \geq b^{\mathrm{dep}}(v')$.

Let E_x denote the set of all transfer edges. Our search operates on the *augmented graph*, which is defined as $G' := (V \cup V_c, E \cup E_x \cup E_c)$. Note that its size is polynomial in the size of G, since the number of dummy vertices of a charging station $v \in S$ is bounded by the number of distinct canonical departure SoC values, which, in turn, is bounded by the number of paths that contribute to profiles from v to other charging stations (or the target vertex).

Algorithm Description. Using the augmented graph, we modify the EVD algorithm introduced in Section 4.2.1 to find energy-optimal routes in the presence of charging stations; see Figure 4.15 for pseudocode. As before, the algorithm takes as input the source $s \in V$, the target $t \in V$, and the initial SoC $b_s \in [0, M]$, but it operates on the augmented graph G'. It maintains a single label $\ell(v)$ per vertex $v \in V \cup V_c$,

```
// initialize labels
foreach $v \in V \cup V_c$ do
    $\ell(v) = (b_v, r_v) \longleftarrow (-\infty, 0)$
$\ell(s) \longleftarrow (b_s, 0)$
$Q$.insert($s, b_s$)
// run main loop
while $Q$.isNotEmpty() do
    $u \longleftarrow Q$.deleteMax()
    $(b_u, r_u) \longleftarrow \ell(u)$
    if $u \in V$ then
        // scan outgoing edges in the original graph and transfer edges
        foreach $(u, v) \in E \cup E_x$ do
            $b \longleftarrow f_{(u,v)}(b_u)$
            $(b_v, r_v) \longleftarrow \ell(v)$
            if $b - r_u > b_v - r_v$ then
                $\ell(v) \longleftarrow (b, r_u)$
                $Q$.update($v, b - r_u$)
    else
        // scan outgoing edges in the charging station graph
        foreach $(u, v) \in E_c$ do
            $b \longleftarrow b^{\text{arr}}(u, v)$
            $r \longleftarrow r_u + b^{\text{dep}}(u) - b_u$
            $(b_v, r_v) \longleftarrow \ell(v)$
            if $b - r > b_v - r_v$ then
                $\ell(v) \longleftarrow (b, r)$
                $Q$.update($v, b - r$)
```

Figure 4.15: Pseudocode of EVD with charging stops. The algorithm expects as input an (augmented) graph $G' := (V \cup V_c, E \cup E_x \cup E_c)$, a cost function $c\colon E \to \mathbb{R}$, a capacity $M \in \mathbb{R}_{\geq 0}$, two mappings $b^{\text{dep}}\colon V_c \to [0, M] \cup \{\infty\}$ and $b^{\text{arr}}\colon E_c \to [0, M]$, a source $s \in V$, a target $t \in V$, and an initial SoC $b_s \in [0, M]$. It computes, for each vertex $v \in V$, the maximum objective $b_v - r_v$ corresponding to the path that minimizes overall consumption.

which stores the best values of SoC $b_v \in [0,M] \cup \{-\infty\}$ and recharged energy $r_v \in \mathbb{R}_{\geq 0}$ of all s–v paths encountered so far, i. e., the pair of values that maximizes the objective $b_v - r_v$ at v. Initially, it sets $b_v = -\infty$ and $r_v = 0$ for all $v \in V$, except for the label $\ell(s) = (b_s, 0)$ at s, which is also inserted into a priority queue. In each iteration of the main loop, the label $\ell(u) = (b_u, r_u)$ of some vertex $u \in V \cup V_c$ in the augmented graph with maximum key $b_u - r_u$ is extracted from the queue. If u is an original vertex, i. e., $u \in V$, the algorithm proceeds exactly like plain EVD by scanning its outgoing edges; see lines 9–14 of the algorithm in Figure 4.15. If, additionally, u is a charging station, i. e., $u \in S$, its corresponding charging vertices $u' \in V_c$ are updated in the priority queue if $b_u < b^{\text{dep}}(u')$ and $\ell(u)$ yields an improvement to the label $\ell(u')$. Note that this is done implicitly in Figure 4.15, by scanning transfer edges and making use of the artificial SoC functions according to Equation 4.3. Alternatively, if $u \in V_c$ is a charging station, it is handled separately by the algorithm; see lines 16–22 in Figure 4.15. All outgoing edges $(u,v) \in E_c$ in G_c are scanned, generating for each a new label (b,r) with $b \in [0,M]$ and $r \in \mathbb{R}_{\geq 0}$ as follows. Its SoC is set to the arrival SoC $b = b^{\text{arr}}(u,v)$ at v. To account for recharging at u, the total amount of charged energy is set to $r = r_u + b^{\text{dep}}(u) - b_u$. If the label (b,r) improves the objective of $\ell(v)$, the latter is updated accordingly and v is inserted or updated in the queue.

After termination, the label at the dummy target vertex t', i. e., the unique vertex $t' \in V_c$ with $b^{\text{dep}}(t') = \infty$, contains the optimal pair of SoC and recharged energy. Correctness of the algorithm follows from the construction of the search graph and the properties of canonical departure SoC discussed above. To retrieve the actual path description, parent pointers are used as in Dijkstra's algorithm (see Section 3.3.1). Underlying paths between vertices $u \in V_c$ and $v \in V_c$ in G_c can be retrieved by (pre)computing an energy-optimal path between their corresponding original vertices with initial SoC $b^{\text{dep}}(u)$.

Complexity. Before we analyze the running time of the modified EVD algorithm, we show that we can make it label setting. We claim that the potential functions described in Section 4.2.1 carry over to the charging station graph G_c, by setting the potential of every vertex in V_c to the potential of its corresponding original vertex. To see this, we define the cost $c(u,v) := b^{\text{dep}}(u) - b^{\text{arr}}(u,v)$ of an edge $(u,v) \in E_c$ as the difference between the objectives before and after scanning the edge (u,v), respectively. Observe that, due to battery constraints, $c(u,v)$ is greater or equal to the cost of a shortest u–v path in the original graph G. Thus, the reduced cost $c(u,v) - \pi(u) + \pi(v)$ is nonnegative for vertex potentials induced by a consistent potential function $\pi\colon V \to \mathbb{R}$ on the original vertices V. The same holds true for transfer edges, since they have nonnegative energy consumption. Thus, we can use any consistent potential function for the original graph to make the algorithm label setting. Note that this also allows us to establish a stopping criterion for the dummy vertex t'.

We argue that the label-setting algorithm enables us to solve the problem of finding energy-optimal routes with charging stops in polynomial time. In summary, it requires the following steps.

1. Compute a consistent potential function for the input graph.
2. Construct the charging station graph G_c.
3. Run our modified EVD algorithm to find an optimal solution.

For the first step, we use the polynomial-time algorithm of Bellman-Ford-Moore [Bel58, For56, Moo59] to compute a vertex-induced potential (see Section 4.2.1). Regarding the second step, we have argued above that the size of the charging station graph G_c is polynomial in the size of the input graph G. To construct it, we have to compute a quadratic number of SoC profiles, which can be done in polynomial time using profile search (see Section 4.2.2). Finally, the third step is solved by the algorithm shown in Figure 4.15. Using potential shifting, it becomes label setting and the number of iterations in the main loop is bounded by the size of the search graph. Hence, an optimal solution is found in polynomial time. Theorem 4.8 summarizes the theoretical insights of this section.

Theorem 4.8. *Given a source $s \in V$ and a target $t \in V$ in the input graph, together with an initial SoC $b_s \in [0, M]$, an energy-optimal s–t path with intermediate charging stops can be computed in polynomial time.*

4.3.4 A Heuristic Implementation

We discuss a practical variant of the EVD algorithm with charging stops, which we introduced in Section 4.3.3. The construction of the subgraph G_c is time consuming on realistic instances. Luckily, we can move most work to preprocessing, since the paths between charging stations are independent of source and target. We also propose a simpler search graph, which can naturally be combined with CH and A* search to achieve further speedup.

First, we obtain vertex potentials during preprocessing, using the more practical variants of EVD described in Section 4.2.1 instead of the algorithm of Bellman-Ford-Moore. Second, we have to construct the charging station graph for a given source $s \in V$ and a given target $t \in V$. In our practical variant, we replace the graph G_c with the overlay $G_S = (V_S, E_S)$, where $V_S := S \cup \{t\}$ and $E_S := S \times (S \cup \{t\})$. Every edge $(u, v) \in E_S$ stores as its cost function the u–v profile (with respect to the original graph). Note that all edges in E_S except for those that have t as their head vertex can be precomputed. Using the overlay G_S instead of G_c has several advantages: It is straightforward to construct G_S using profile search, and the number of vertices in

```
   // initialize labels as in Figure 4.15
   // run main loop
1  while $Q$.isNotEmpty() do
2      $u \longleftarrow Q$.deleteMax()
3      $(b_u, r_u) \longleftarrow \ell(u)$
4      if $u \in S$ then
           // scan shortcuts between charging stations
5          foreach $v \in S \cup \{t\}$ do
6              $b^* \longleftarrow \arg\max_{b \in [\max\{b_u, b_u^{\min}\}, \max\{b_u, b_u^{\max}\}] \cup \{b_u\}} f_{(u,v)}(b) - b$
7              $r \longleftarrow r_u + b^* - b_u$
8              $b \longleftarrow f_{(u,v)}(b^*)$
9              $(b_v, r_v) \longleftarrow \ell(v)$
10             if $b - r > b_v - r_v$ then
11                 $\ell(v) \longleftarrow (b, r)$
12                 $Q$.update($v, b - r$)
13     else
           // scan outgoing edges in the original graph as in Figure 4.15
```

Figure 4.16: A heuristic variant of EVD with charging stops. It takes an input graph $G = (V, E)$, a cost function $c \colon E \to \mathbb{R}$, a source $s \in V$, a target $t \in V$, a battery capacity $M \in \mathbb{R}_{\geq 0}$, and the initial SoC $b_s \in [0, M]$. Moreover, it requires a set $S \subseteq V$ of charging stations with specified charging ranges. The algorithm computes an SoC $b_v \in [0, M] \cup \{-\infty\}$ and a corresponding amount $r_v \in \mathbb{R}$ of charged energy for each $v \in V$.

the search graph is significantly smaller. Additionally, integration with CH (described below) becomes much simpler.

Shortcuts $(u, v) \in E_S$ in the overlay G_S are used during the search to greedily determine the departure SoC at the charging station $u \in V_S$ and the arrival SoC at the vertex $v \in V_S$. This requires a slight modification to the EVD algorithm; see Figure 4.16 for pseudocode. During its main loop, the next vertex $u \in V$ with label $\ell(u) = (b_u, r_u)$ is determined as before. If u represents a charging station, i. e., $u \in S$, all outgoing shortcuts $(u, v) \in E_S$ are scanned. For each, the departure SoC $b^* \in [0, M]$ that maximizes the objective $f_{(u,v)}(b^*) - (r_u + b^* - b_u)$ at v is picked under the constraint that $b^* \geq b_u$ and b^* lies in the charging range of u (the case $b^* = b_u$ is always allowed, to account for the possibility of not recharging at u); see line 6 in Figure 4.16. If this yields an improvement to the label at v, it is updated accordingly. Making use of vertex potentials, the search becomes label setting and stops as soon as the target vertex is extracted from the priority queue.

As argued before, picking the SoC at v in this greedy fashion may lead to suboptimal results. For example, in Figure 4.13 the upper path between u and v (with consumption 0) always maximizes the objective at v, but the bottom path (with consumption 2) is the better choice for low initial SoC, as it requires less charging and enables recuperation on the subsequent v–t path. On real-world networks, however, this is very unlikely to occur, as it requires an optimal route with two charging stops $u \in S$ and $v \in S$, such that the target t can be reached from u via v, but *not* directly, whereas charging too much energy at u (to reach v on an optimal s–v path) prevents recuperation along the v–t path due to a fully charged battery. Consequently, our heuristic approach *always* produced optimal solutions in our tests; see Section 4.5.2.

Integration with Contraction Hierarchies. To enable faster queries, we propose CH [Gei+12b], which have been extended to EV scenarios before [EFS11]. As in basic CH (see Section 3.3.2), we iteratively *contract* vertices in increasing order of (heuristic) importance during preprocessing, maintaining distances between all remaining vertices by adding *shortcut* edges, if necessary. Similar to previous approaches [Sto12a], we do not contract vertices that represent charging stations. Hence, we contract only some vertices, which form the *component*. This leaves an uncontracted *core*, which is an overlay graph that contains all charging stations (and possibly other vertices). Note that in our scenario, a shortcut (u, v) corresponds to a u–v profile, so shortcuts must store SoC functions. Shortcuts are computed and updated during vertex contraction, using the general link and merge operation described in Section 4.1.3. Consequently, SoC functions with multiple breakpoints may emerge during contraction, which makes preprocessing more expensive. After contraction has stopped, we run profile searches on the (relatively small) core graph to quickly compute shortcuts between charging stations and construct the overlay G_S. Shortcuts are only added to G_S if their corresponding SoC function is *finite* for some SoC (i. e., the head vertex is reachable from the tail).

In a basic approach, witness search of CH preprocessing (c. f. Section 3.3.2) is replaced by profile search to determine whether a shortcut is necessary. For faster preprocessing, an alternative variant uses only the *maximum finite consumption* of an edge e with SoC function f_e in the current overlay graph, i. e., the finite value $c_e^{\max \leq M} := \max_{b \in [0,M]}\{b - f_e(b) \mid f_e(b) \neq -\infty\}$ that maximizes its energy consumption. Observe that negative costs are ruled out this way, since consumption must be at least 0 for a fully charged battery. Hence, the witness search operates on a graph with scalar, nonnegative costs. This reenables Dijkstra's algorithm, which then computes an upper bound $\bar{c} \in \mathbb{R}_{\geq 0}$ on the energy consumption between a given pair of vertices. A shortcut candidate is inserted only if its SoC function f consumes less energy for some SoC, i. e., there exists an SoC $b \in [0, M]$ with $b - f(b) < \bar{c}$. Using these upper bounds, we may end up inserting unnecessary shortcuts. This does not affect correctness, but may slow down queries slightly. (Similarly, Eisner et al. use a sampling approach to avoid costly profile search during preprocessing in their implementation [EFS11].)

In summary, our preprocessing routine comprises three steps: (1) computation of a consistent potential function, (2) CH preprocessing, and (3) construction of the overlay G_S. Afterwards, the query algorithm runs in two phases. The first runs a profile search from the target $t \in V$ on the backward graph of the component and the core, which contain original edges and shortcuts computed during preprocessing. In the component, the search scans only *upward* edges with respect to the vertex order (c. f. Section 3.3.2). Shortcuts between charging stations in G_S are ignored by this search. After its termination, SoC profiles from each charging station to the target are known. We (temporarily) add the target and all corresponding shortcuts to the overlay graph G_S. Similarly, we include a (temporary) shortcut from any vertex $v \in V \setminus S$ visited by the profile search to the target. Then, the second phase runs the modified variant of EVD sketched in Figure 4.15 from the source $s \in V$ with initial SoC $b_s \in [0, M]$ on a search graph consisting of upward edges in the component and all edges in the core (including G_S).

To obtain the full path description, we enable path unpacking by storing via vertices during contraction, as in plain CH [Gei+12b]. Note, however, that we need one via vertex per contributing path of an SoC function. Additionally, we have to reconstruct paths represented by shortcuts between charging stations within the core. This can be done by precomputing and storing the paths in the core graph explicitly, or by running an EVD search on the core graph between each consecutive pair of charging stations in the path. Finally, the paths in the core are unpacked as in plain CH. The amount of energy that must be recharged at a charging station is easily obtained from the SoC profiles in the overlay G_S, by picking a departure SoC $b \in [0, M]$ for each profile f that maximizes the objective $f(b) - b$ at the next station.

Incorporating A* Search. On instances with many charging stations, scanning shortcuts in the dense overlay graph G_S induced by all charging stations becomes the major bottleneck of the search. To reduce the search space, we combine our approach with A* search [HNR68]. The basic idea of A* search is to change the order in which vertices are extracted from the priority queue, such that vertices closer to the target are extracted first (see Section 3.3.2).

Prior to the forward search of a query, we run Dijkstra's algorithm from the target vertex on the backward graph, scanning upward edges in the component and all core edges, except for shortcuts between charging stations. The algorithm uses *minimum energy consumption* as edge costs, defined for an edge e with SoC function f_e in the search graph as $c_e^{\min} = \min_{b \in [0,M]} b - f_e(b)$. Since energy consumption can become negative, the search is label correcting unless potential shifting is applied. After its termination, each vertex label stores a scalar *lower bound* on the energy consumption that is necessary to reach the target from this vertex. This yields a vertex-induced potential for all vertices in the core (c. f. Section 4.2.1).

The forward search is then split into two phases. The first runs from the source $s \in V$ in the component, using potentials computed during preprocessing. It is *pruned* at core vertices, i. e., the algorithm scans no outgoing edges from these vertices. The second phase runs on the core graph enriched with the overlay G_S, including shortcuts to the target $t \in V$. The search is initialized with all core vertices scanned in the first phase, but uses potentials obtained by the backward search. As the potential of each vertex is a lower bound on energy consumption on the way to t, the second phase is goal directed (vertices closer to the target have smaller keys).

An aggressive variant of A* search achieves further speedup at the cost of suboptimal results. As before, when a charging station $u \in S$ is visited by the forward search, all outgoing shortcuts $(u, v) \in E_S$ in G_S are scanned. However, we update the label of at most *one* vertex $v \in S$ and insert it into priority queue, namely, the tentative label with maximum key among all vertices that are improved by the scans. The subgraph G_S is rather dense, so this significantly reduces the number of subsequent vertex scans.

Implementation Details. During contraction, we determine the next vertex that is contracted using the measures Edge Difference (ED) and Cost of Queries (CQ) according to Geisberger et al. [Gei+12b]. The rank of a vertex is then set to $4\,\mathrm{ED} + \mathrm{CQ}$ (recall that vertices of lower rank are contracted first). To improve query times, we reorder vertices after preprocessing, such that core vertices are in consecutive memory for improved locality. During CH queries, the forward EVD search and the backward profile search are executed alternately, as in plain CH. Thereby, we avoid an exhaustive run of the costly profile search in cases where the target is close to the source.

4.4 Extending Customizable Route Planning

Energy consumption values in the input graph may vary with, e. g., vehicle load and changing weather conditions, or due to newly learned consumption data [GM17, Mas+14]. This makes fast preprocessing particularly important in our context. We introduce a speedup technique that focuses on fast integration of changes in the cost function. It extends the CRP approach introduced by Delling et al. [Del+17], which exploits ideas from MLD [Del+09, HSW09, JP02, SWW00, SWZ02]. The algorithm has three phases: a (potentially costly) offline metric-independent *preprocessing phase*, a *customization phase* that handles metric-dependent preprocessing, and the (online) *query phase*. Its main strength is that customization is very quick: A new cost function can be incorporated in a few seconds, even on continental networks, while a single edge cost can be updated in only a few microseconds [Del+17]. In this section, we tackle the problem setting introduced in Section 4.1.1. Hence, we do not consider stops at charging stations. In the following, we recap the preprocessing phase (Section 4.4.1) and the query phase (Section 4.4.2) of the MLD algorithm, describing our extensions along the way.

4.4.1 Preprocessing and Customization

The preprocessing phase computes a multilevel overlay [JP02, SWZ02] of the input graph $G = (V,E)$. It is obtained from a nested multilevel partition $\Pi = (\mathcal{V}^1, \dots, \mathcal{V}^L)$ of the vertices of G as follows. For a fixed level $\ell \in \{1, \dots, L\}$, the overlay graph of level ℓ contains all edges of G that are boundary edges of $\mathcal{V}^\ell$. Moreover, there is a shortcut edge (u,v) for every pair $u \in V_i^\ell$ and $v \in V_i^\ell$ of boundary vertices per cell $V_i^\ell \in \mathcal{V}^\ell$. This results in a full clique of edges over a cell's boundary vertices; see Section 3.3.2 for details. As preprocessing is metric independent, no changes have to be made to adapt it to our scenario. Moreover, it only needs to be rerun if the *topology* of the input changes (significantly). Since this happens infrequently in practice, somewhat higher preprocessing times are no issue.

Customization. The customization phase uses the output of the preprocessing phase to compute the metric[1] of the overlays, i. e., for each shortcut edge it must compute its SoC function. It proceeds in a bottom-up fashion, starting with the lowest level 1. Within a fixed level $\ell \in \{1, \dots, L\}$, each cell $V_i^\ell \in \mathcal{V}^\ell$ is processed independently. A cell V_i^ℓ is processed by running, for each boundary vertex $u \in V_i^\ell$, a profile search from u restricted to the subgraph induced by V_i^ℓ (i. e., it does not relax any edges pointing outside V_i^ℓ). At every boundary vertex $v \in V_i^\ell$, this results in a u–v profile, which is assigned to the clique edge (u,v) of V_i^ℓ. Note that when processing a level $\ell \in \{2, \dots, L\}$, we make use of the already computed overlay graph of level $\ell - 1$ by running the profile search on this overlay, which improves customization time significantly.

Parallelization. Customization can be parallelized by distributing different cells (on the same level) among processors. In contrast to scalar costs in plain CRP, the complexity of SoC functions is not known in advance. Thus, our overlay uses a single dynamic adjacency array to store breakpoints of shortcut edges. Note that updates to this data structure must be synchronized. A common approach is using locks, which is costly. Instead, each thread locally maintains a log of the SoC functions it has computed. The logs are merged sequentially after processing each level.

Preliminary experiments indicated that more than 80 % of the functions have simple form, so they can be compressed to constant size (see Section 4.1.3). Only for the remaining cases a thread uses its log, while compressed functions are written to the

[1]Formally, energy consumption does not define a metric on the input graph, due to negative costs and the lack of symmetry. Nevertheless, we stick to the term as it is commonly used in the literature.

(preallocated) overlay directly. Unlike the preprocessing phase, customization is much faster, taking mere seconds in practice when executed in parallel.

Implementation Details. Similar to Delling et al. [Del+17], we use a compact representation to store the overlays: Instead of keeping separate graphs, we store a common vertex set for *all* levels, which is equivalent to the set of boundary vertices of $\mathcal{V}^1$. Only shortcut edges are kept in a separate data structure per level, and they are organized as matrices of preallocated contiguous memory (note that boundary edges are already present in the input graph). Each matrix entry comes with a flag to indicate whether it stores a compressed function or an index in the array containing the breakpoints of the corresponding SoC function, similar to the dynamic adjacency array used during profile search (see Section 4.2.2).

As vertices of the input graph G are represented by indices $\{1, \ldots, n\}$ in our implementation, we can reorder them such that overlay vertices of higher levels are pushed to the front, breaking ties by cell index. Non-overlay vertices are ordered by their level-1 cell indices. This improves spatial locality for both customization and queries, and simplifies mapping between original and overlay vertices.

When running profile searches during customization, a naïve implementation constructs a label per vertex of the input graph. Exploiting that search graphs are limited to cells of the partition, we can save a significant amount of space by reducing the number of distinct vertex labels. After reordering vertices during preprocessing, we compute the range of vertex indices per cell and level. During customization, we can remap the ranges of each level of the current cell to a smaller range of indices. The length of this range depends on the maximum cell size in any of the overlays, which is known after preprocessing. The following (mixed) variant worked best in our experiments: We only remap bottom-level vertex indices of non-boundary vertices (the majority of vertices), while keeping a distinct vertex label for each boundary vertex of any cell in the graph. Thereby, we save a significant amount of space and improve locality, but keep vertex mapping overhead limited during customization. Note that only customization on the lowest level is affected by remapping in this variant.

To quickly reset labels of overlay vertices between different profile searches, we do not resort to standard approaches like timestamps, but exploit once more that vertices are reordered. We explicitly reset the labels of all overlay vertices contained in the current cell (on the current level). With labels of each level being on a contiguous range of memory, this can be done efficiently in practice.

Finally, we use *clique flags* [Bau+16f] to reduce the number of edge scans during profile searches. For each vertex $v \in V$ in the overlay, a flag indicates if for *any* parent vertex $u \in V$ (i. e., any second to last vertex on a contributing path of the current profile in the label of v), it holds that (u, v) is a boundary edge. Only if the flag is set, we relax outgoing clique edges of v when it is scanned. Note that this does not violate correctness, as there always exists an optimal path in the overlay that does not contain two consecutive clique edges. This follows immediately from the triangle inequality and the fact that we use full cliques in the overlay.

4.4.2 Queries

For a source $s \in V$, a target $t \in V$, and an SoC $b_s \in [0, M]$, the query operates on a search graph G' consisting of (1) the overlay graph of the topmost level L, (2) all cell-induced subgraphs in the overlays of all levels that contain s or t, and (3) the subgraphs of the original graph induced by the level-1 cells that contain s or t. Then, any algorithm described in Section 4.2 can be run on this search graph to get provably optimal solutions for both query types. Also, potentials computed for the original graph naturally carry over to the overlays. Therefore, we assume that potentials are available in the remainder of this section. We refer to EVD running on the search graph specified above as *Unidirectional MLD (Uni-MLD)*. Similarly, we refer to profile search as *Profile-MLD* when run on this search graph. Note that the search graph does not need to be constructed explicitly. Instead, the level and cell on which Uni-MLD or Profile-MLD scan edges are determined implicitly from the partition data [Del+17]. Just as in plain MLD, shortcut edges e at level $\ell \in \{1, \ldots, L\}$ can be unpacked to obtain the full path description after t was reached, by (recursively) running a local query on the overlay of level $\ell - 1$, restricted to the level-ℓ cell containing e (recall that level 0 corresponds to the original graph).

In what follows, we discuss techniques to accelerate SoC queries based on *bidirectional search* [Dan63, Del+17]. Basically, bidirectional search simultaneously runs a *forward search* from s on G' and a *backward search* from t on $\bar{G}'$ until a stopping condition is met. Observe that, given a consistent potential function π on G', we immediately obtain a consistent potential $\bar{\pi}$ on the backward graph $\bar{G}'$ by setting $\bar{\pi}(v) := -\pi(v)$ for all $v \in V$. The algorithm maximizes a *tentative SoC value* $b^* \in [0, M] \cup \{-\infty\}$ (initialized to $-\infty$) whenever the searches meet at some vertex $v \in V$. After stopping, the shortest path with target SoC b^* is obtained (if it exists) by concatenating the corresponding s–v path and v–t path found by the searches. Unfortunately, the final SoC at t is not known in advance, which prevents running a regular backward EVD search. Instead, we present two approaches that augment the backward search [DW09]. We denote them by *Bidirectional Profile-Evaluating MLD (BPE-MLD)* and *Bidirectional Distance-Bounding MLD (BDB-MLD)*. Both use a regular forward EVD search.

Bidirectional Profile-Evaluating MLD. The first approach, BPE-MLD, runs a *backward profile search* from t, which does not require an initial SoC value. It computes SoC functions f_v representing v–t profiles for all $v \in V$ as vertex labels. Whenever the forward or backward search scans an edge toward a vertex $v \in V$ that has already been touched by the opposite search, it evaluates the SoC function f_v (obtained from the backward search) for the SoC $b := b(v)$ (obtained from the forward search) and updates $b^* = \max\{b^*, f_v(b)\}$. The algorithm may stop as soon as *any* path it may still find has SoC below b^*. Recall from Section 4.2.2 that the profile search uses minimum energy consumption of a vertex label (plus a potential) as key in its priority queue. Let k_F denote the current *maximum* key in the queue of the forward search and let k_B denote the current *minimum* key in the queue of the backward search. Then we can stop the search as soon as the condition $k_F - k_B \leq \max\{b^*, 0\}$ holds.

Bidirectional Distance-Bounding MLD. Unfortunately, running a backward profile search can be costly. Therefore, the second approach, BDB-MLD, runs a cheaper backward search that *bounds* the forward search in order to "guide" it toward t. (Note that a similar idea is used by Gutman [Gut04].) However, we have to carefully account for battery constraints. To do so, the backward search maintains three labels for each vertex $v \in V$, namely, a lower and an upper bound on the cost of an energy-optimal path from v to t, denoted $\underline{c}(v)$ and $\bar{c}(v)$, and an upper bound on the minimum SoC that is necessary to reach t, denoted $\bar{b}(v)$. We define $\bar{c}(v)$ consistently with $\bar{b}(v)$: An SoC of $\bar{b}(v)$ implies that t can be reached from v with cost at most $\bar{c}(v)$. Labels are initially set to ∞, except at t, for which they are set to $\underline{c}(t) = \bar{c}(t) = \bar{b}(t) = 0$. The backward search then runs Dijkstra's algorithm based on the labels $\underline{c}(\cdot)$ (we use potential shifting to ensure that the search is label setting). When scanning an edge $e = (u, v)$ in the backward graph $\bar{G}'$, it uses its minimum energy consumption $c_e^{\min} = \min_{b \in [0,M]} b - f_e(b)$ as edge cost. During the same edge scan, $\bar{c}$ and $\bar{b}$ are computed as follows. Let $b_e^{\min} \in [0, M]$ denote the minimum SoC that is necessary to traverse e, i. e., the smallest SoC value for which f_e is finite. Then the bound on the minimum SoC $\bar{b}(v)$ to travel from v to t (via u) is determined by the maximum of $b_e^{\min}$ itself and the sum of the cost $c_e(b_e^{\min}) = b_e^{\min} - f_e(b_e^{\min})$ of traversing e with SoC $b_e^{\min}$ plus $\bar{b}(u)$, the minimum SoC to get from u to t. On the other hand, the maximum cost $\bar{c}(v)$ at v is determined by $\bar{c}(u) + c_e^{\max \leq M}$, where $c_e^{\max \leq M}$ is the *maximum finite consumption* of the edge, i. e., the finite value that maximizes $c_e(b) = b - f_e(b)$ for $b \in [0, M]$. Summarizing, whenever the algorithm scans some edge $e = (u, v)$ in the backward search graph, it checks whether $\max\{b_e^{\min}, c_e(b_e^{\min}) + \bar{b}(u)\} \leq \bar{b}(v)$ and $\bar{c}(u) + c_e^{\max \leq M} \leq \bar{c}(v)$, updating $\bar{b}(v)$ and $\bar{c}(v)$ if necessary.

The tentative SoC value b^* is now maintained by the forward search and corresponds to a lower bound on the target SoC of the energy-optimal s–t path, initialized to $-\infty$. Whenever it scans a vertex $v \in V$ with SoC label $b(v)$ that was already visited by the backward search, it checks if $b(v) \geq \bar{b}(v)$. Only in this case, it tries to update b^* by setting $b^* = \max\{b^*, b(v) - \bar{c}(v)\}$. Moreover, given the current keys k_F and k_B of the forward and backward search, respectively, the following test is performed (independently of the previous check). The search is *pruned* at v (i. e., edges outgoing from v are not scanned), if either v was already settled by the backward search and $b(v) - \underline{c}(v) \leq \max\{b^*, 0\}$, or v was not settled by the backward search and $b(v) - k_B \leq \max\{b^*, 0\}$ holds. The algorithm stops when the forward search reaches t and determines the SoC $b(t)$. We stop the backward search early if $k_F - k_B \leq 0$ holds.

Parallelization. To get additional speedup, we propose parallelizing the search in a multi-core scenario. We assign different processors to the forward and backward search, where they run independently. To update the tentative SoC value $b^* \in [0, M] \cup \{-\infty\}$, each search must access vertex labels of the opposite search, potentially involving a race condition. However, as long as reads to vertex labels are atomic, race conditions can safely be ignored: The correct value b^* will always be determined by the opposite search at a later point. Unfortunately, the backward search of BPE-MLD maintains non-atomic functions as vertex labels. Updating b^* is therefore restricted to the backward search (accesses to labels of the forward search are still atomic). To ensure correctness, the forward search checks, whenever it scans a vertex $v \in V$, if v has already been touched by the backward search (which is an atomic read). If so, it adds v to a list. At the end, this list is processed sequentially, checking if any vertex labels improve b^*. Note that this list is small in practice.

Reachability Flags. If the target vertex is not reachable from the source (with the given initial SoC), the forward search simply visits all reachable vertices, while the backward search visits all vertices from which the target can be reached with at least some initial SoC. To quickly identify and accelerate long-distance queries for which the target is unreachable, we can additionally precompute *reachability flags*: For the topmost level L of the partition, we keep a bit matrix, whose entry (i, j) is set if the cell V_j^L is *reachable* from any vertex in cell V_i^L (with a full battery). To set the matrix entries during customization, we run, for each cell V_i^L at level L (in parallel), a multi-source variant of Dijkstra's algorithm on the level-L overlay from all boundary vertices of the cell V_i^L. (One could also interpret the search as Dijkstra's algorithm running from a super source that is added to the overlay together with edges to all boundary vertices with energy consumption 0.) It uses lower bounds $\min_{b \in [0,M]} b - f_e(b)$ on consumption as cost of a given edge e in the overlay. We set all flags (i, j) of the matrix for which there exists a boundary vertex of cell V_j^L at distance at most M. In practice, storing these bits requires little additional space. During a query, we first check the flag for the pair of cells containing s and t. If it is not set, we may stop immediately.

Implementation Details. We reuse several techniques from the customization to further improve queries. In particular, we exploit again that vertices are represented by indices $\{1, \ldots, n\}$ and reordered during preprocessing. Recall that we precompute and store the corresponding range of vertex indices for each cell and level. After a query, we reset only the labels of the (at most two) cells per level containing s and t, along with all labels of vertices on the level-L overlay. We also use clique flags during queries. Note that this becomes even simpler for SoC queries compared to profile search, since parent vertices are always unique in this case (c. f. Section 4.4.1).

Additionally, we save space by storing cell indices (for each level) only at the *boundary* vertices of a cell. Before running the actual query algorithm, indices of the source and target cell (which are required to implicitly construct the search graph) are retrieved at negligible overhead by running a depth-first search (DFS) [Cor+09] from both the source and the target, until each encounter a boundary vertex.

4.5 Experiments

We evaluate all our approaches on our main benchmark instance Eur-PTV. We use two different EV models. The first, denoted PG-16, is based on the PHEM model of a Peugeot iOn and has a battery capacity of 16 kWh. The second, EV-85, is based on the artificial PHEM model, for which we assume a larger battery capacity of 85 kWh (as in high-end Tesla models). See Section 3.4 for details on input data and methodology. Below, we evaluate our basic algorithms (Section 4.5.1), approaches for routes with charging stops (Section 4.5.2), and our customizable technique (Section 4.5.3). Finally, we also compare our new algorithms to previous approaches (Section 4.5.4). Unless noted otherwise, we ran our implementation sequentially on machine-s. All reported query times are average values of 1 000 queries. SoC queries assume a full battery, i. e., $b_s = M$ at the vertex source $s \in V$. Note that a lower initial SoC would only result in faster query times.

4.5.1 Basic Algorithms

We evaluate the variants of EVD and profile search discussed in Section 4.2. Since the range of the vehicle models PG-16 and EV-85 is restricted, evaluating random queries (as it is common) would not be meaningful: For most queries, the target vertex would be unreachable. Further, in most cases we can easily identify such out-of-range queries with little effort, e. g., by utilizing reachability flags; recall Section 4.4.2. Instead, we generate *in-range* queries by first picking a random source vertex $s \in V$ uniformly at random, from which we run a preliminary search with initial SoC $b_s = M$. The target vertex $t \in V$ is picked uniformly at random from its search space (i. e., all vertices within the vehicle's range).

Evaluating Queries. Table 4.1 reports figures for our basic algorithms and 1 000 random in-range queries. In addition to basic EVD, we ran the same queries after establishing a *global* stopping criterion (denoted sc-g, using the minimum cost c^* of any path in the graph for the stopping criterion) and a *local* stopping criterion (sc-ℓ,

Table 4.1: Evaluation of basic algorithms on Eur-PTV for both vehicle models. For different variants of the EVD algorithm (employing different stopping criteria) and for each vehicle model, we report space consumption in bytes per vertex (B/n), customization time, as well as the average number of vertex scans and running time of queries.

	PG-16				EV-85			
	Custom.		Query		Custom.		Query	
Algorithm	Space [B/n]	Time [s]	# Vertex Scans	Time [ms]	Space [B/n]	Time [s]	# Vertex Scans	Time [ms]
EVD	—	—	388 817	49.6	—	—	4 392 002	636.5
EVD-sc-g	0.0	2.61	321 728	40.8	0.0	3.44	2 823 076	402.8
EVD-sc-ℓ	4.0	2.61	196 704	24.6	4.0	3.44	2 284 079	323.5
EVD-π_v	4.0	3.43	184 322	18.3	4.0	3.48	2 089 126	219.6
EVD-π_b	4.0	2.61	184 164	23.0	4.0	3.44	2 137 157	295.0
EVD-π_h	4.0	0.37	184 523	21.9	4.0	0.37	2 137 282	292.2
Profile-π_h	4.0	0.37	192 559	31.0	4.0	0.37	2 212 806	410.6

storing the value c_v^* for every $v \in V$). We also tested the different potential functions (π_v, π_b, and π_h). See Section 4.2.1 for details on the different stopping criteria. Except for the basic variant, metric-dependent preprocessing is required if edge costs change. For EVD-sc-g and EVD-sc-ℓ, customization times indicate the time to compute the values c_v^* for each $v \in V$. For all other variants, we show the time required to compute the potential function. The space overhead is exactly four bytes (one integer) per vertex for all variants with a stopping criterion, except EVD-sc-g (which only keeps one integer in total).

Regarding queries, we report the number of vertex (re)scans and query times. It is not surprising that the basic label-correcting approach is the slowest for both models, although the number of vertex *rescans* is comparatively low (less than 10 % of all vertex scans are rescans of vertices that were already scanned before; not shown in the table). With EVD-sc-g, we achieve a first speedup, but the rather weak stopping criterion still results in a large search space size. Nevertheless, observe that computing the offset c^* already amortizes after 15 queries on average for EV-85, while producing virtually no space overhead. The local stopping criterion (EVD-sc-ℓ) yields another improvement in running time at the cost of higher space consumption. The different variants of vertex potentials allow for even better query times by making EVD label setting. Somewhat surprisingly, EVD-π_v yields the best times, however, with higher variance (available from Figure 4.17). Since EVD-π_h is more robust and provides the best customization time by far, we use height-induced potentials for profile search and all algorithms tested in the remainder of this chapter. Finally, we see that—in contrast to time-dependent route planning [Bat+13, Del11, DW09]—profile queries

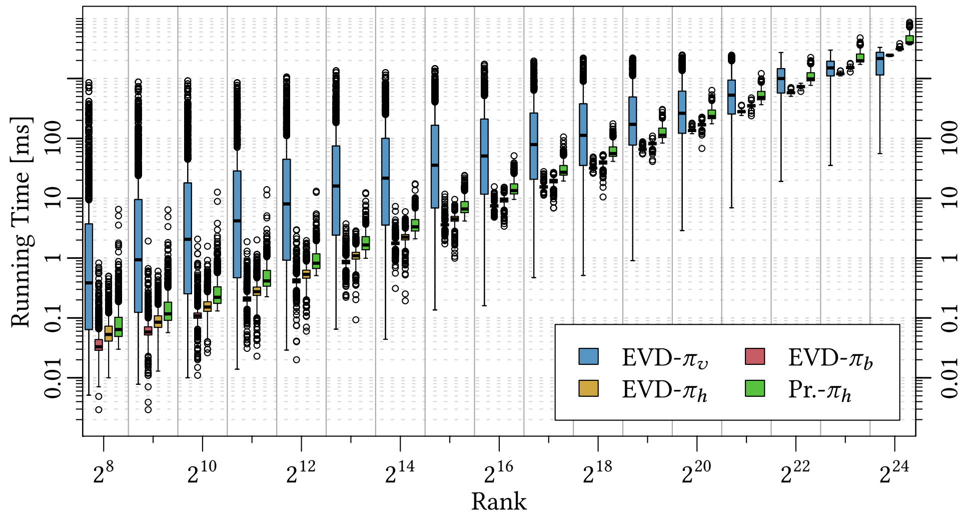


Figure 4.17: Running times of basic algorithms subject to Dijkstra rank (EV-85). Low ranks indicate local queries. Battery capacity is increased to the point where range is not constrained.

admit practical running times in our scenario: We observe a slowdown by a factor of less than 2 compared to EVD.

Evaluating Scalability. We analyze the scalability of our basic algorithms for the EV-85 model, following the *Dijkstra rank* method [Bas+16, SS05]. Given two vertices $s \in V$ and $t \in V$, the Dijkstra rank with respect to s and t is the number of vertex scans performed by Dijkstra's algorithm in an s–t query, presuming that the algorithm stops as soon as t is scanned. Thus, higher ranks reflect harder queries. Given that costs can be negative in our scenario, the label-correcting variants of EVD may scan vertices multiple times, while vertex potentials strongly influence the order in which vertices are scanned. Hence, we use a slightly altered definition of Dijkstra rank in this chapter: We order the vertices by the time they were last extracted from the priority queue when running label-correcting EVD and determine ranks from this order. As in regular Dijkstra ranks, the maximum rank is bounded by the graph size. For each rank in $\{2^1, \dots, 2^{\lfloor \log n \rfloor}\}$, we generated 1 000 queries this way from sources chosen uniformly at random. To get meaningful results, we increased the battery capacity from 85 kWh to 1 000 kWh, which corresponds to a cruising range of roughly 5 000 km—enough to make the target reachable in all queries.

Figure 4.17 shows resulting query times for EVD and profile search with different potential functions in a box-and-whisker plot. Interestingly, EVD-π_v has much higher variance compared to EVD-π_b and EVD-π_h. Moreover, query times of EVD-π_v are lower for long-distance queries, but significantly worse for local queries of low rank.

Recall that the potential function π_v is induced by distances from a single vertex, which results in a highly distorted search space. Apparently, the lower average query time of EVD-π_v reported in Table 4.1 is mostly induced by long-distance queries, whereas more local queries are typically rather costly. For the highest ranks, median running times of all approaches are above two seconds. Profile search is consistently slower than EVD (except EVD-π_v for low ranks) by a factor of at most 2–3.

Remarks. In conclusion, only a label-correcting variant of EVD that does not employ a stopping criterion can be used without preprocessing effort. Using a global value for the stopping criterion (EVD-sc-g) offers mild speedup at negligible space consumption. All other methods require an additional integer value to be stored with each vertex. Potential functions offer polynomial guarantees on running time and are slightly faster in practice. Fastest average query times are achieved by EVD-π_v after a few seconds of customization. An alternative variant, EVD-π_b, yields more robust query times and slightly faster customization. Finally, the potential function π_h requires that elevation data of the network is available and consistent with consumption data. Yet, it offers the lowest customization time (less than 0.5 seconds) and robust query times that compete with the other techniques. Furthermore, note that we include space overhead of four bytes per vertex for storing the height-induced potential $\pi_h(v) = \alpha \cdot h(v)$ at each vertex $v \in V$. If height values are already available as part of the input, we may as well just store the single value $\alpha \in \mathbb{R}$, and compute $\pi_h(v)$ on demand in the algorithm. This reduces space overhead to a *single* integer value (similar to EVD-sc-g).

4.5.2 Routes with Charging Stops

To analyze our algorithms that allow intermediate stops at charging stations, we conducted experiments on Eur-PTV and the subnetwork Ger-PTV representing Germany. Unless mentioned otherwise, we use the 13 810 charging stations (1 966 of them in Germany) located on ChargeMap (see Section 3.4). All charging stations have the SoC range $[0, M]$. As before, the initial SoC in each query is $b_s = M$. Reported query times are average values of 1 000 queries, with source and target vertices picked uniformly at random. All algorithms use height-induced potential functions.

Evaluating Queries. Table 4.2 evaluates performance of CH for different core sizes. In this experiment, vertex contraction on Ger-PTV was stopped as soon as the average degree of vertices in the core reached a given threshold. Although it is possible to contract all vertices except for charging stations at moderate preprocessing effort (less than 15 minutes), we observe that aborting contraction earlier actually improves query times. This is due to the fact that the number of shortcuts (and hence, the number of edge scans during queries) is much smaller when using a larger but also sparser core graph. Consequently, query times of CH are fastest for an average core degree of 48,

Table 4.2: Impact of core size on performance (Ger-PTV, PG-16). Vertex contraction stopped once the average degree in the core reached a given threshold (Ø Deg.), or only charging stations were left in the core. We report resulting core size (# Vertices), preprocessing time, and average query times for CH as well as CH combined with A* search.

Core size		Prepr.	Query [ms]	
Ø Deg.	# Vertices	T. [s]	CH	CH+A*
0	—	176.8	526.3	1 681.8
16	31 063 (0.66 %)	257.5	16.3	16.9
32	5 904 (0.13 %)	416.7	12.0	4.9
48	3 472 (0.07 %)	548.5	11.2	4.2
64	2 701 (0.06 %)	633.7	11.8	4.1
128	2 029 (0.04 %)	786.8	12.6	6.4
∞	1 966 (0.04 %)	832.7	12.4	8.9

while CH combined with A* search achieves best results for an average degree of 64. Compared to a variant that does not contract any vertices and only computes the charging station graph G_S (first row of Table 4.2), this results in a speedup by a factor of almost 45 for CH and 410 for CH combined with A* search. Note that A* search does not pay off for large core sizes, as the backward profile search becomes a bottleneck. In all following experiments that involve CH, we stop contraction on Ger-PTV at an average core degree of 48. On Eur-PTV, we set this threshold to 32 (obtained in preliminary experiments).

Table 4.3 compares different approaches to compute energy-optimal routes with charging stops on our main test instance (Eur-PTV), for both vehicle models. Applied techniques are indicated by the four leftmost columns. The first row (no speedup technique enabled) shows our exact baseline approach introduced in Section 4.3.2. It requires no preprocessing, but takes 20–30 seconds to answer queries, which is rather impractical. Simply plugging in the charging station graph G_S and using the modified EVD (see Section 4.3.4) already reduces query times significantly. However, scalability of this approach is limited, because increasing the vehicle range affects both preprocessing (longer paths between charging stations must be precomputed) and queries (the search in the uncontracted network dominates running times). Integrating CH clearly pays off, as it further reduces the number of vertex scans and query time after a moderate preprocessing effort of less than an hour. Query times of CH are dominated by the search in G_S. A* search helps reducing the effort spent searching in G_S and makes our approach rather practical, with running times of less than 300 ms for the artificial model. Even though we use a formally inexact implementation, the optimal solution is found in *all* queries.

The aggressive variant of A* search further reduces query times at the cost of inexact results, even in practice. The average relative error (not reported in the table) is 0.7 %

Table 4.3: Performance of approaches taking charging stops into account (Eur-PTV). Columns G_S, CH, A*, and Ag. (Aggressive A*) indicate whether a technique is enabled (●) or not (○). For each approach and model, we report preprocessing time, the number of vertex scans during queries (# V. Sc.), and query times.

Techniques				PG-16			EV-85		
G_S	CH	A*	Ag.	Prepr. [s]	# V. Sc.	T. [ms]	Prepr. [s]	# V. Sc.	T. [ms]
○	○	○	○	—	8 895 038	20 160.9	—	11 033 760	32 928.8
●	○	○	○	1 487	759 951	710.0	15 062	7 753 601	6 285.7
●	●	○	○	2 860	8 433	309.6	3 246	19 616	1 281.5
●	●	●	○	2 860	3 563	128.2	3 246	10 418	297.5
●	●	●	●	2 860	1 599	41.0	3 246	9 579	157.8

for PG-16 and less than 0.01 % for the artificial EV-85 model. This discrepancy in relative error can be explained by the fact that a larger battery allows the EV to stick to energy-optimal paths (fewer detours are necessary), so the quality of the bounds used in A* search increases. Consequently, outliers for PG-16 exceed 10 % in relative error in about 1 % of the cases, while even the maximum error is below 0.5 % for EV-85. For all techniques, queries for EV-85 are harder to solve. This is mostly due to the dense charging station graph (in case of the baseline approach, more labels created per vertex), since more charging stations are reachable from each station.

Evaluating Scalability. Running times of all approaches are dominated by the search in the charging station graph. Hence, we analyze the effect of different types of charging stations, the total number of stations, and vehicle range on overall performance. In Table 4.4, we evaluate the performance of our fastest empirically exact approach (CH combined with A* search) under varying types and distributions of charging stations. We consider five different scenarios.

The first scenario (reg-cm) uses stations from ChargeMap with default SoC ranges $R_v = [0, M]$ for all charging stations $v \in S$. The second (mix-cm) uses the same stations, but assigns to each vertex $v \in S$ the charging range of a regular station ($R_v = [0, M]$), a supercharger that quickly charges to 80 % SoC ($R_v = [0, 0.8M]$), or a swapping station ($R_v = [M, M]$), with equal probability. The results indicate that SoC ranges have little effect on performance. This is not surprising, since restricting the departure SoC can only reduce the search space (the effect is negligible, though).

Furthermore, we consider random distributions of charging stations (reg-r0.01, reg-r0.1, reg-r1.0) with default SoC ranges, where we pick 0.01 %, 0.1 %, and 1.0 % of the vertices in V as charging stations, respectively, chosen uniformly at random. We observe that the number of charging stations has a more significant impact on algorithm performance. Given that the number of edges in G_S grows quadratically

Table 4.4: Performance for varying distributions of charging stations (Ger-PTV, PG-16). We investigate our fastest empirically exact approach (CH+A*). Besides timings for preprocessing and queries, we report the number of charging stations ($|S|$), edges in G_S ($|E_S|$), as well as the average number of vertex scans (# V. Sc.) and edge scans (# E. Sc.).

		Prepr.		Queries		
Scenario	$|S|$	T. [s]	$|E_S|$	# V. Sc.	# E. Sc.	T. [ms]
reg-cm	1 966	548.5	539 145	4 592	125 535	4.22
mix-cm	1 966	548.1	539 145	4 592	125 381	4.19
reg-r0.01	469	487.2	22 231	2 234	50 070	1.30
reg-r0.1	4 692	582.7	2 263 310	8 904	223 779	7.97
reg-r1.0	46 920	965.0	227 514 459	60 527	1 828 581	73.46

in the number of charging stations, preprocessing and queries slow down for very dense charging networks. This limits scalability, but our approach handles realistic distributions of charging stations (note that for the scenario reg-1.0, the number of charging stations is larger than the current number of gas stations in Germany).

Figure 4.18 shows running times of our algorithms for different battery capacities. We use the PG-16 model, but vary its battery capacity as indicated in the plot. Without A* search, running times roughly double with battery capacity, because G_S becomes denser and hence, the number of reachable charging stations increases. Adding A* search, scalability improves significantly, since vertex potentials quickly guide the search towards the target and decrease the search space in the dense subgraph G_S.

4.5.3 Customizable Energy-Optimal Routes

Since our implementation of MLD exploits parallelism in both metric-dependent preprocessing and queries, experiments in this section were conducted on machine-p. As partitioning tool we used PUNCH (Partitioning Using Natural Cut Heuristics) [Del+11b], which is explicitly developed for road networks and aims at minimizing the number of boundary edges. Given a bound $\bar{k} \in \mathbb{N}$, it partitions the vertices of the input graph G into cells with at most $\bar{k}$ vertices each. We proceed by first partitioning the topmost level. Lower levels are computed by recursively running PUNCH on each cell-induced subgraph (of a higher level) independently. For Europe, we use a 4-level partition with maximum cell sizes 2^6, 2^{10}, 2^{14}, and 2^{18}, respectively (values determined in preliminary experiments). Computing the partition took 24 minutes. Considering that the road topology rarely changes (the partition needs to be updated only when roads are built or closed), this is sufficiently fast in practice.

Evaluating Queries. Table 4.1 reports figures for our MLD algorithms on the main test instance Eur-PTV, using the models PG-16 and EV-85 and the same set of 1 000

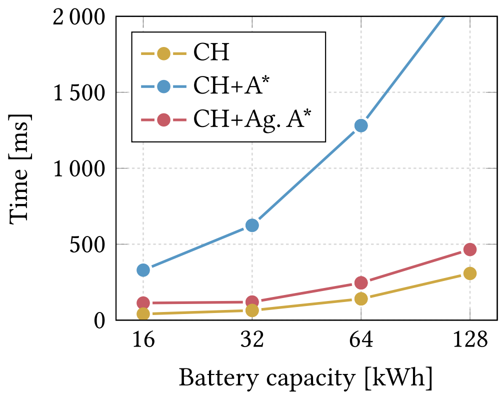


Figure 4.18: Running times subject to cruising range (Eur-PTV, PG-16). Each point in the plot corresponds to the median running time of 1 000 queries for one of the different approaches (CH, CH with A*, CH with aggressive A*) and varying battery capacities.

queries as in Section 4.5.1. Recall that the target is always reachable in these queries. Customization times include both metric customization and potential computation (we do not use reachability flags). For comparison, we also show results for EVD. All algorithms use height-induced potentials. We also parallelize the computation of the potential function, although the achieved speedup is moderate (factor of 3–4).

Regarding MLD, we see that customization takes less than four seconds when parallelized, enabling frequent metric updates for the whole network. When executed sequentially, customization takes 34.8 seconds (respective 40.4 seconds) for the PG-16 (EV-85) model (not reported in the table). Thus, parallelization on 16 cores yields a very good speedup factor of about 11 in both cases. Customization of a single cell, e. g., when only local updates are required, is much faster and takes about 100 ms (not shown in the table). In all cases, customization times for EV-85 are higher than for PG-16. We attribute this to the larger number of negative edges in the former instance (see Section 3.4).

Space consumption is dominated by breakpoints of profiles, which are piecewise linear functions. Most profiles (about 80 %) have compressed form, so they are stored as a single 32-bit integer. Of the remaining profiles, the majority (more than 90 %) consist of at most two breakpoints. For both models, there are only very few (below 2 000 out of over 30 million) shortcuts with profiles containing 10 or more breakpoints. As a result, overhead in space consumption is moderate, requiring only a few bytes per vertex. One can further reduce it by removing the lowest level of the partition for the query phase, keeping it only to accelerate customization [Del+17]: For both models, this saves space by a factor of 2, while queries are slowed down by only about 10 % on average (not reported in the table). Furthermore, note that for all variants, we include space overhead for storing the height-induced potential at each vertex. As mentioned in Section 4.5.1, we can save space by just keeping a single value $\alpha \in \mathbb{R}$ for the whole graph. Altogether, taking these measures can reduce customization space to about four bytes per vertex for each model.

Table 4.5: Evaluation of MLD approaches for both vehicle models (Eur-PTV). We report figures as in Table 4.1, for the same set of 1 000 queries.

	PG-16				EV-85			
	Custom.		Query		Custom.		Query	
Algorithm	Space [B/n]	Time [s]	# Vertex Scans	Time [ms]	Space [B/n]	Time [s]	# Vertex Scans	Time [ms]
EVD	4.0	0.19	184 523	27.48	4.0	0.19	2 137 282	369.19
Uni-MLD	13.6	3.20	900	0.37	14.5	3.67	2 305	1.24
BPE-MLD	13.6	3.20	891	0.30	14.5	3.67	2 194	0.92
BDB-MLD	13.6	3.20	1 120	0.25	14.5	3.67	2 754	0.67
Pr.-MLD	13.6	3.20	1 068	0.75	14.5	3.67	2 763	3.60

All MLD query variants provide SoC query times of below 2 ms, for both vehicle models. Compared to EVD, this improves query times by more than two orders of magnitude. Bidirectional search also clearly outperforms Uni-MLD. We observe that BDB-MLD is faster than BPE-MLD by about 20–30 % on average. Note, however, that depending on the application, bidirectional search might not pay off: It is run on two cores, but the speedup achieved is (slightly) less than 2. Finally, our approach also enables profile queries within a few milliseconds (Pr.-MLD). Compared to unidirectional SoC queries, we observe a slowdown by a factor of 2–3 on average.

Evaluating Scalability. Figure 4.19 shows scalability of our MLD algorithms, using the Dijkstra rank method as explained in Section 4.5.1. For the same set of 1 000 random queries per rank, we report results for Uni-MLD, BPE-MLD, BDB-MLD, and profile search (Pr.-MLD). As before, we use the EV-85 model, but set battery capacity to 1 000 kWh. We observe that except for very local queries (below rank 2^{12}), bidirectional search always pays off. Moreover, our most sophisticated method for SoC queries, BDB-MLD, is consistently the fastest approach for all ranks. Using BDB-MLD, we achieve maximum query times of under 4.0 ms for the highest ranks, while Uni-MLD stays below 6.4 ms. Profile search, on the other hand, is slightly slower for all ranks and produces most outliers. This can be explained by the fact that running times vary with the number of breakpoints necessary to represent profiles. Thus, times may increase for mountainous areas, where profiles are likely to consist of more breakpoints and shortcut scans become particularly expensive. As a result, we obtain maximum profile query times of over 70 ms. Nevertheless, using MLD yields a speedup of more than two orders of magnitude compared to plain profile search.

Figure 4.20 shows running times subject to Dijkstra rank for the PG-16 model. Again, we use the same set of queries as in Figure 4.17. However, in contrast to Figure 4.19, we

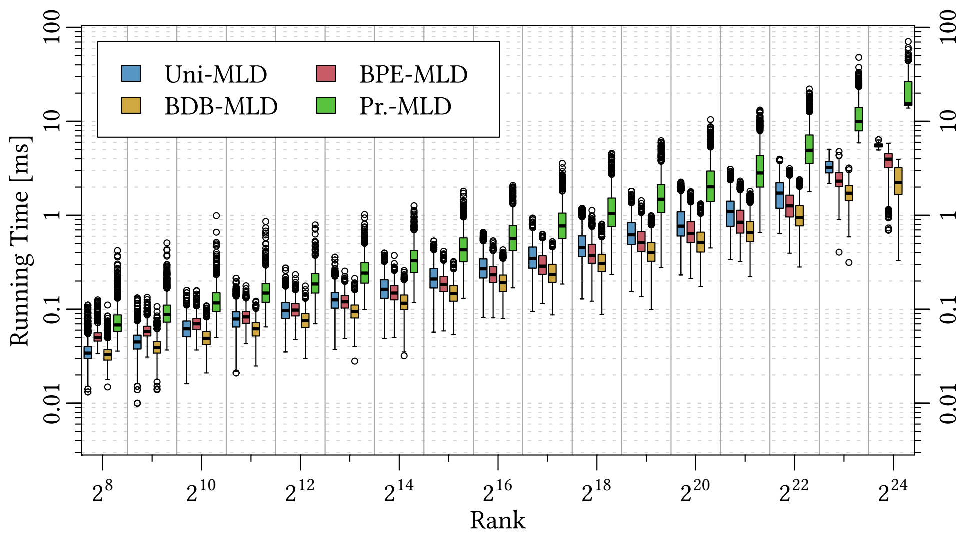


Figure 4.19: Running times of MLD subject to Dijkstra rank. We use the same vehicle model and queries as in Figure 4.17. Battery capacity is increased such that range is not constrained.

enable reachability flags and keep battery capacity at 16 kWh. Hence, the plot shows the effect of reachability flags on long-distance queries for different MLD variants. Starting with rank 2^{18}, query times drop gradually. Beyond rank 2^{22}, the target is almost never reachable, which results in median query times of under 0.01 ms for Uni-MLD. Differences in query times between the techniques for high ranks are explained by initialization overhead, which is more expensive for variants that employ profile search (because dynamic data structures have to be cleared). Similar to Figure 4.19, BDB-MLD is consistently the fastest approach except for very high ranks, where queries are always aborted after initialization, while profile search is the slowest algorithm. The topmost level of our partition contains 99 cells, hence, reachability flags require 99^2 bits (less than 10 kb) of space in total. Computing them in parallel took less than 5 ms.

4.5.4 Comparison of Approaches

We compare the performance of different approaches that compute routes for EVs (without charging stops). First, we consider the fastest previous speedup technique to solve the problem and the new approaches presented in this work. Second, we examine energy consumption on paths that minimize travel time or distance.

Comparison of Speedup Techniques. At the time of writing, the fastest available approach that computes energy-optimal routes for EVs is based on CH [EFS11, Sto13]. The authors adapt plain CH [Gei+12b] to the scenario of optimizing energy consump-

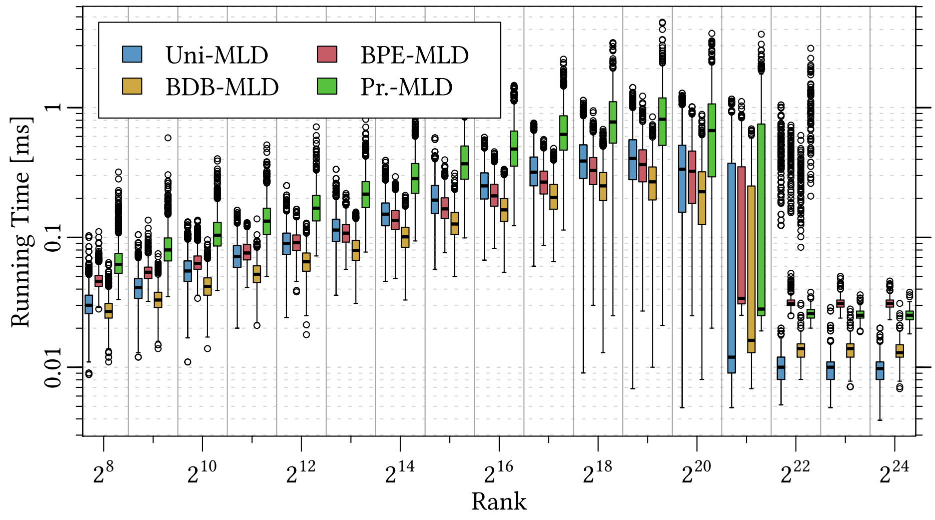


Figure 4.20: Running times of the PG-16 model subject to Dijkstra rank, with reachability flags enabled. As in Figure 4.17, lower ranks indicate more local queries.

tion in the following way: To avoid costly profile computation in witness searches during preprocessing, they acquire upper bounds on witness paths by sampling. This simplifies preprocessing, but may result in a larger number of shortcuts. For the bidirectional CH query, they extract the whole backward search graph with a BFS instead of running a profile search.

To compare our MLD algorithms and our own implementation of CH for EVs with the existing method, we ran experiments on the largest instance used in the previous works, Jap-OSM [Sto13], which was kindly given to us by the authors. The instance is based on an OSM export of the road network of Japan, augmented with SRTM data. It has 26 million vertices and 54 million edges; see Table 3.1 in Section 3.4. Note that these figures are slightly higher than for our main instance (Eur-PTV), however, OSM networks are notorious for having exceptionally many vertices of low degree that only model geometry. Taking this into account, our MLD approach uses a 4-level partition with increased maximum cell sizes of 2^7, 2^{11}, 2^{15}, and 2^{19} vertices, respectively. Using PUNCH [Del+11b], computing the partition took less than half an hour. Eisner et al. [EFS11, Sto13] use the simple consumption model stated in Equation 3.1 in Section 3.4, which is based on the geographical distance and height difference of edges. Note that a height-induced potential follows from the model (we set $\alpha := -\mu$). Therefore, computing the potential function does not require any customization time. Similar to Eisner et al. [EFS11, Sto13], we assume a very large battery capacity. As a result, the target is always in range and reachability flags are disabled in our algorithm.

Table 4.6: Comparison of speedup techniques for SoC queries (Jap-OSM). For different variants and implementations of EVD, CH, and MLD, we report space consumption (in bytes per vertex) and time for preprocessing. For queries, we report the average number of vertex scans and timings. For figures taken from existing work [Sto13], we also report scaled timings.

	Custom.		Query	
Algorithm	Space [B/n]	Time [s]	# Vertex Scans	Time [ms]
EVD [Sto13]	4.0	—	14 431 809	6 492.58
EVD [our]	4.0	—	12 661 423	2 044.63
CH [Sto13]	23.0	14 329.87	10 024	44.93
CH (scal.) [Sto13]	23.0	7 188.77	10 024	14.15
CH [our]	44.8	1 076.74	252	0.88
Uni-MLD	7.7	1.83	2 196	0.67
BPE-MLD	7.7	1.83	2 252	0.62
BDB-MLD	7.7	1.83	2 650	0.46

Table 4.6 reports results on Jap-OSM for our implementation of CH following the description in Section 4.3.4, but contracting *all* vertices in the graph because there are no charging stations ($S = \emptyset$), as well as different variants of MLD. The experiments were conducted on machine-p. Additionally, the table shows figures for existing implementations of EVD and CH [Sto13]. Since they were obtained on slower machines, we report scaled timings. There are two established approaches to scale running times between machines: (1) Using running times of a common baseline algorithm, (2) having access to the same hardware for scaling experiments. Eisner et al. [EFS11, Sto13] use two machines (an AMD Opteron 6172 with 2.1 GHz for preprocessing, an Intel i3-2310M with 2.1 GHz for queries), so we resort to both scaling approaches. For query times of CH, we obtain a scaling factor based on the EVD implementations, maintaining their speedup factor of about 145. Since we have an Opteron 6172 available, scaling of preprocessing time is done by our own scaling experiment. Although not specifically mentioned, we infer that the existing EVD implementation [EFS11, Sto13] uses a stopping criterion: The reported search space is about 56 % of the graph size.

At first glance, Jap-OSM seems to be harder than Eur-PTV: Our EVD variant scans more vertices and has higher query times on Jap-OSM, due to the larger graph size and unlimited range. However, we observe that all MLD variants perform better on Jap-OSM than on Eur-PTV. Observe that the modeling overhead in OSM has an impact only on the lowest level of the partition. Regarding CH, our implementation is significantly faster in both preprocessing and queries, but has higher space consumption compared to the existing variant [EFS11, Sto13]. The latter can be explained by the fact that,

Table 4.7: Comparison of energy-optimal routes to other metrics. For routes that minimize travel time or distance, respectively, we report the percentage of routes that become infeasible (Unr.), the additional amount of energy spent, and the loss in the respective metric (travel time or distance) when using an energy-optimal path.

	Travel Time			Distance		
Instance	Unr.	Extra Energy	Extra Time	Unr.	Extra Energy	Extra Dist.
Eur-PTV (PG-16)	56 %	41 %	47 %	23 %	11 %	5 %
Eur-PTV (EV-85)	62 %	61 %	62 %	28 %	16 %	4 %
Jap-OSM	—	—	—	0 %	25 %	11 %

unlike Eisner et al., we contract *all* vertices of the graph and maintain via vertices for *every* breakpoint of profiles (which is simple but also redundant). However, contracting all vertices clearly pays off in terms of query performance: The average search space is smaller by a factor of 40 compared to the existing implementation. Interestingly, MLD provides the best query times. At the same time, its (metric-dependent) preprocessing is faster than CH by more than a factor of 500 and requires a fraction of the space. Even when run on a single core, customization of MLD still only requires 19.6 seconds and is more than 50 times as fast as CH preprocessing. Our findings add to previous observations that, compared to CH, separator-based approaches are more robust towards metrics other than (unconstrained) travel time [Bau+16f, Del+17, DSW16]. Moreover, CH suffers from its bidirectional nature, since the backward profile search becomes the major bottleneck of SoC queries. Consequently, CH outperforms MLD when answering profile queries (not reported in the table). In this case, average query times of CH are only slightly higher (0.98 ms), while MLD is slowed down by a factor of 3 (1.83 ms). Thereby, our techniques also outperform a previous implementation of profile search based on CH by Schönfelder et al. [SLW14] (they report an average query time of 19 ms on a much smaller graph and mention that their implementation is not finely tuned).

Comparison of Metrics. We also compare energy-optimal routes to those that minimize travel time and covered distance, respectively. Table 4.7 shows results for Eur-PTV and Jap-OSM. We use the same 1 000 queries as in Table 4.1 for Eur-PTV and Table 4.6 for Jap-OSM, respectively. For each metric, we report the percentage of queries where the target becomes unreachable when optimizing travel time or distance. For cases where the target is reachable, we show the average amount of extra energy spent on the quickest or shortest route (instead of the energy-optimal one) and the extra time or distance required on the energy-optimal route. Travel times were not available for Jap-OSM, so we only evaluate the distance metric on this instance.

As driving speed has a huge impact on energy consumption, minimizing the travel time greatly reduces range. Consequently, more than half of the targets that are reachable on an energy-optimal route become unreachable when taking the quickest route instead. Even if the target is reachable on both routes, optimizing one criterion greatly increases the other. This effect becomes less significant when comparing energy consumption to distance. This indicates that there is a strong correlation between energy consumption and covered distance. However, since there are many other factors—such as road type and slope—that influence energy consumption, minimizing travel distance still fails to retain reachability of the target in more than 20 % of the cases on Eur-PTV. In conclusion, explicitly optimizing for energy consumption clearly pays off and increases the range of an EV significantly.

4.6 Final Remarks

We studied the computation of energy-optimal routes for EVs. Key challenges included negative costs to model recuperation and battery capacity constraints. We examined SoC profiles that capture these constraints and proved that their complexity is at most linear in the graph size. Furthermore, we derived basic algorithms to solve two relevant query types, namely, SoC queries and profile queries. We investigated different strategies to establish stopping criteria and developed a polynomial-time algorithm for profile queries.

We also discussed energy-optimal routes with charging stops and showed how profile search can be utilized to solve the problem in polynomial time. The problem setting can be seen as a transition between (efficiently solvable) energy-optimal routes without charging stops [EFS11, Sac+11] and $\mathcal{NP}$-hard time-constrained variants that include charging stops [SMS17] (which generalize the problem setting considered in this chapter). Our findings prove that it is indeed the addition of a second optimization criterion (travel time) that makes the latter settings $\mathcal{NP}$-hard, rather than the incorporation of charging stations in combination with battery constraints; see also Section 5.2. We also proposed a practical variant, which (empirically) computes optimal results in well below a second on realistic, large-scale networks.

Finally, we presented algorithms based on the CRP approach, which in addition to the above challenges, handle frequently changing metrics in a sound manner. We integrated profile search into customization and discussed a nontrivial adaptation of bidirectional search. On the continental network of Europe, our approach incorporates new metrics within seconds and answers queries in less than a millisecond—making it the fastest available technique for energy-optimal routing of EVs.

Future Work. Next steps include the integration of turn costs (in terms of energy consumption), where recuperation due to braking must be taken into account [Har12].

Realistic models of turn costs are important to produce meaningful results in practice, as energy-optimal routes often resort to minor roads comprising many turns. Regarding routes with charging stops, interesting lines of future work include reducing the number of edges in the overlay of charging stations for better performance and scalability of CH [Del+17, HSW09, SWZ02] or integration with CCH [DSW16] for faster preprocessing. It might also be worthwhile to extend the proposed A* search to an adaptation of ALT [GH05, GW05] to enable faster queries. Additionally, one could consider a *profile* variant of this problem setting, i. e., ask for an SoC profile with intermediate charging stops. We are also interested in including further ideas on tuning the CRP approach [Del+17, DW13]. Moreover, note that as described in this chapter, customization has to be rerun whenever the battery capacity of a vehicle changes. However, custom capacities may be desirable in many situations, e. g., when modeling battery aging or user constraints on minimum SoC during a ride. Therefore, one could make use of a more flexible representation of profiles that is *independent* of the capacity $M \in \mathbb{R}_{\geq 0}$ (e. g., by explicitly storing lengths of certain important subpaths of contributing paths; see the characterization of SoC profiles given by Lemma 4.1 in Section 4.1.2). Then, the parameter M could be part of the query input.

5 Shortest Feasible Paths for Battery Electric Vehicles

In Chapter 4, we developed algorithms to compute routes for EVs that minimize energy consumption. It turned out that all considered problem variants allow efficient solutions, both in theory and in practice. However, our evaluation also revealed that—compared to fastest routes—routes that solely optimize for energy consumption typically come with a great increase in travel time and vise versa; recall Table 4.7 in Section 4.5. The reason for this is that driving slowly reduces aerodynamic drag, which has a major impact on energy consumption [Bed+16, FAR16, HF14, LL12]. As a result, energy-optimal routes often contain disproportionate detours using slow roads, which users may only be willing to accept in some cases, e. g., when the battery is nearly depleted. Most of the time, however, users are interested in routes that keep energy consumption low, but still allow the driver to reach the target within a reasonable time frame. Trading travel time for energy consumption inherently results in a bicriteria problem, on which we set our focus in this chapter.

We consider variants of the $\mathcal{NP}$-hard Constrained Shortest Path (CSP) problem that capture specific requirements of EVs. We show that fastest routes subject to battery constraints can be computed by a straightforward adaptation of the (exponential-time) multicriteria shortest path algorithm. Based on this insight, we consider two nontrivial extensions of the basic problem. First, we incorporate stops at *charging stations*. We take into account that the charging process is nonlinear and can be interrupted to reach the target earlier. Using realistic and flexible models of charging stations, our approach allows different types of charging stations to be present in the network, such as stations with varying charging power or battery swapping stations. Second, we take into account that in reality, travel time and energy consumption are not only affected by the choice of the route itself, but also by driving behavior. Hence, it might pay off to save energy by deliberately driving below posted speed limits, especially along high-speed roads. Careful speed planning becomes even more relevant with the advent of autonomous vehicles, where driving speeds can be planned in advance to ensure that the target is reached [Flo+15]. We discuss different ways to model such *adaptive speeds* and propose algorithmic solutions to solve the extended problem setting. Since all problem variants considered in this chapter are $\mathcal{NP}$-hard, we do not guarantee polynomial running times of our exact algorithms. Instead, we demonstrate their practicality on realistic input in an extensive evaluation. We also propose heuristic variants, the fastest of which compute high-quality solutions within tens of milliseconds on large-scale networks, as our experimental study reveals.

Chapter Overview. In Section 5.1, we state the basic problem setting, which is an extension of the well-known $\mathcal{NP}$-hard CSP problem. We also describe a baseline algorithm that solves it in exponential time, by adapting the multicriteria shortest path algorithm [Han80, Mar84] presented in Section 3.3.1.

In Section 5.2, we extend the basic problem to planning routes that, while respecting battery constraints, minimize overall trip time, including time spent at charging stations. Our solution handles all types of charging stations accurately: battery swapping stations, regular charging stations with varying charging power, as well as superchargers that quickly charge to a certain fraction of the maximum SoC. In particular, charging times are *not* independent of the SoC when arriving at a charging station in our model. Additionally, the charging process can be interrupted as soon as further charging would delay the arrival at the target. We first show how the basic algorithm can be extended to handle charging stops. Carefully incorporating recharging models in speedup techniques based on A* search and CH, we are able to solve the problem optimally within reasonable time on realistic instances. For faster queries, we propose heuristic approaches that offer high (empirical) quality.

In Section 5.3, we study a generalization of the CSP problem that takes adaptive speeds into account. Using realistic consumption models, we obtain a function for each road segment that maps driving time to energy consumption. This results in a challenging problem, which as a first step, we solve with an extension of the basic algorithm. To reduce practical running times, we incorporate techniques based on A* search and CH, which can be combined for further speedup. One of the most challenging aspects in this setting is the computation of shortcuts, which represent bivariate functions in our model. We also discuss heuristic variants.

Section 5.4 presents our comprehensive experimental study on detailed and realistic data. For both considered problem settings, it demonstrates that we can compute *optimal* solutions within seconds and below for realistic query scenarios and within minutes or less for long-distance queries, on par or faster than previous *heuristic* algorithms. Thereby, our approaches outperform the state-of-the-art, even though they are designed to solve more complex problems. Using our heuristic variants, we achieve query times that are fast enough for interactive applications, while providing high-quality solutions. We conclude this chapter in Section 5.5 with a summary and an outlook on interesting future work.

5.1 Basic Problem Setting

In the basic variant of our problem, we are given a graph $G = (V, E)$ together with two cost functions $d\colon E \to \mathbb{R}_{\geq 0}$ and $c\colon E \to \mathbb{R}$ representing *driving time* and *energy consumption* on an edge, respectively. As before, consumption values can become negative to model recuperation, but the battery capacity $M \in \mathbb{R}_{\geq 0}$ imposes constraints

```
// initialize label sets
foreach $v \in V$ do
    $L(v) \longleftarrow \emptyset$
$L(s) \longleftarrow \{(0,b_s)\}$
$Q$.insert($(0,b_s),s,0$)
// run main loop
while $Q$.isNotEmpty() do
    $(\ell = (x,b),u) \longleftarrow Q$.minElement()
    foreach $(u,v) \in E$ do
        $x' \longleftarrow x + d(u,v)$
        $b' \longleftarrow \min\{M, b - c(u,v)\}$
        if $b' \geq 0$ then
            $\ell' \longleftarrow (x',b')$
            if $L(v)$ does not dominate $\ell'$ then
                $L(v)$.deleteLabelsDominatedBy($\ell'$)
                $L(v)$.insert($\ell'$)
                $Q$.update($\ell',v,x'$)
```

Figure 5.1: Pseudocode of the label-setting BSP algorithm for EVs. It takes as input a graph $G = (V,E)$ with cost functions $d\colon E \to \mathbb{R}_{\geq 0}$ and $c\colon E \to \mathbb{R}$, a source $s \in V$, a target $t \in V$, a battery capacity $M \in \mathbb{R}_{\geq 0}$, and an initial SoC $b_s \in [0,M]$. It computes the minimum (constrained) driving time from s to t for the initial SoC b_s.

on the feasibility of paths; see Section 4.1.1. For a query consisting of a source $s \in V$, a target $t \in V$, and an initial SoC $b_s \in [0,M]$, we seek to compute a path that is feasible and minimizes driving time. If all consumption values are nonnegative, this problem is equivalent to the well-known $\mathcal{NP}$-hard CSP problem [HZ80], which (in the terminology of our setting) asks for a path with minimum driving time such that energy consumption does not exceed the threshold M. Consequently, our problem at hand is $\mathcal{NP}$-hard as well.

To solve the problem, we can adapt the (exponential-time) multicriteria shortest path algorithm [Han80, Mar84] from Section 3.3.1 in a straightforward manner. See Figure 5.1 for pseudocode of the resulting *bicriteria shortest path (BSP)* algorithm for EVs. It maintains label sets, in our case containing tuples (x,b) of driving time $x \in \mathbb{R}_{\geq 0}$ and SoC $b \in \mathbb{R}$. A label (x,b) *dominates* another label (x',b') if $x \leq x'$ and $b \geq b'$. Initially, all label sets are empty, except for the label $(0,b_s)$ at the source s, which is also inserted into the priority queue. In each step, the algorithm extracts the label $\ell = (x,b)$ with minimum driving time $x \in \mathbb{R}_{\geq 0}$, assigned to some vertex $u \in V$. We also say that ℓ is *settled* at this point. The algorithm then scans all edges $(u,v) \in E$ outgoing from u. If the new label $\ell' := (x + d(u,v), b - c(u,v))$ is not dominated by any label in $L(v)$, it is added to $L(v)$ and the queue, removing labels dominated by ℓ' from $L(v)$. Battery constraints can be incorporated on-the-fly by additional checks during the algorithm as follows. When scanning an edge $(u,v) \in E$, we set the SoC of the new label ℓ' to $\min\{M, b - c(u,v)\}$. If this results in negative SoC, we discard the label ℓ'. Using driving time as key of tuples in the priority queue (breaking ties by SoC, i. e., giving preference to the label with highest SoC if two or more labels have the same driving time), the algorithm is *label setting*, i. e., extracted labels are never dominated afterwards. An optimal (constrained) solution is then found once the first label at the target t is settled. As in the basic algorithm from Section 3.3.1, parent pointers can be added to labels to retrieve the optimal path itself.

Note that, while we focus on the problem of computing fastest feasible paths, it is not hard to adapt all algorithms introduced in this chapter to closely related problem settings, such as

- computing a path with minimum driving time such that the target is reached with at least a certain minimum SoC;
- computing a path with minimum driving time such that the SoC does not fall below a certain threshold at *any* point during a journey;
- computing the energy-optimal path that does not exceed a certain travel time;
- computing the full Pareto set of nondominated solutions at the target (with respect to driving time and energy consumption).

In the remainder of this chapter, we generalize the CSP problem to complex, realistic problem settings in the context of route planning for EVs. In particular, we consider stops at charging stations and adaptive speeds.

5.2 Integrating Charging Stops

Besides a limited cruising range, a major difference between traveling with EVs and their conventional counterparts running on combustion engines is that charging stations are still much rarer than gas stations and recharging is time consuming. Thus, routes can become infeasible without intermediate charging stops (the battery runs empty) and fast routes may be less favorable when taking longer recharging times into account. In this section, we aim to find the overall fastest route, considering battery constraints and charging stops when necessary. For that, our model has to integrate different kinds of charging stations in a sound manner, such as battery swapping stations and stations with varying charging power. Moreover, charging time depends on the SoC when arriving at a charging station. The charging process can also be interrupted as soon as further charging would delay the arrival at the target. Note that this stands in contrast to the problem considered in Section 4.5.2, where we only optimized energy consumption and ignored travel time.

In what follows, we formally specify our model and the problem (Section 5.2.1). Afterwards, we derive a basic approach, which extends the (exponential-time) BSP algorithm (Section 5.2.2). It turns out that as its most crucial ingredient, new labels must be generated at charging stations to model battery charging. We discuss how a limited number of labels can be constructed to ensure termination and correctness of the approach (Section 5.2.3). To improve practical performance of our exponential-time algorithm, we propose tuning based on A* search (Section 5.2.4) and the speedup technique CH (Section 5.2.5). We also discuss how both techniques can be combined to further reduce practical running times (Section 5.2.6). Finally, we introduce heuristic approaches, which drop correctness for additional speedup (Section 5.2.7).

5.2.1 Model and Problem Statement

As in the basic problem setting (see Section 5.1), we assume that we are given a graph $G = (V, E)$ with two edge cost functions $d \colon E \to \mathbb{R}_{\geq 0}$ and $c \colon E \to \mathbb{R}$ representing driving time and energy consumption, respectively. Additionally, a subset $S \subseteq V$ of the vertices represents charging stations. We allow stops at charging stations to recharge the battery while spending charging time. Each vertex $v \in S$ has a designated *charging function* $\mathrm{cf}_v \colon [0, M] \times \mathbb{R}_{\geq 0} \to [0, M]$, which maps *arrival SoC* and the spent charging time to the resulting *departure SoC*. We presume that charging functions are continuous and monotonically increasing with respect to charging time (i. e., charging for a longer time never decreases the SoC). Further, we assume that for arbitrary charging times $x_1 \in \mathbb{R}_{\geq 0}$, $x_2 \in \mathbb{R}_{\geq 0}$ and SoC values $b \in [0, M]$, the *shifting property* $\mathrm{cf}_v(\mathrm{cf}_v(b, x_1), x_2) = \mathrm{cf}_v(b, x_1 + x_2)$ holds. Hence, charging speed only depends on the current SoC, but not on the arrival SoC. These conditions are met by realistic physical models of charging stations [Mon+17, Pel+17, Uhr+15]. Moreover, exploiting the shifting property, it is possible to represent the (bivariate) charging function cf_v using a univariate function $\tilde{\mathrm{cf}}_v \colon \mathbb{R}_{\geq 0} \to [0, M]$ with $\tilde{\mathrm{cf}}_v(x) := \mathrm{cf}_v(0, x)$; see Figure 5.2 and our explanation further below.

Given a vertex $v \in S$ with a charging function cf_v that has the above properties, we further presume there is a finite value $x_v^{\max} \in \mathbb{R}_{\geq 0}$, such that $\mathrm{cf}_v(b, x) = \mathrm{cf}_v(b, x_v^{\max})$ for all $b \in [0, M]$ and $x \geq x_v^{\max}$. In other words, some maximum SoC is reached after a finite charging time (the charging function does not converge to some SoC without reaching it eventually). Then, the minimum SoC value $b_v^{\min} := \min_{x \in \mathbb{R}_{\geq 0}} \mathrm{cf}_v(0, x) = \mathrm{cf}_v(0, 0)$ and the maximum SoC value $b_v^{\max} := \max_{x \in \mathbb{R}_{\geq 0}} \mathrm{cf}_v(0, x) = \mathrm{cf}_v(0, x_v^{\max})$ of a charging function induce a range $[b_v^{\min}, b_v^{\max}]$ of possible SoC values after charging at v. We allow the cases $b_v^{\min} > 0$ and $b_v^{\max} < M$ to model certain restrictions of charging stations. For example, we include swapping stations by setting $\mathrm{cf}_v(x, b) = M$ for all values $x \in \mathbb{R}_{\geq 0}$ and $b \in [0, M]$. Hence, we obtain $b_v^{\min} = b_v^{\max} = M$. Our notion

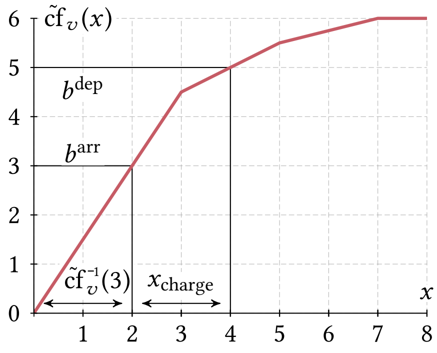


Figure 5.2: A univariate charging function $\tilde{\mathrm{cf}}_v$ of a charging station $v \in S$ with minimum SoC value 0 and maximum SoC value 6. Reaching v with an arrival SoC of $b^{\mathrm{arr}} = 3$ and spending a charging time of $x_{\mathrm{charge}} = 2$ yields a departure SoC $b^{\mathrm{dep}} = \mathrm{cf}_v(b^{\mathrm{arr}}, x_{\mathrm{charge}})$, which we obtain by evaluating $\tilde{\mathrm{cf}}_v(\tilde{\mathrm{cf}}_v^{-1}(b^{\mathrm{arr}}) + x_{\mathrm{charge}}) = 5$.

of charging functions generalizes the definition given in Section 4.3.1 and is flexible enough to capture features of realistic charging stations. Finally, we assign to every charging station $v \in S$ a constant *initialization time* $x_{\mathrm{init}}(v)$ that is spent when charging at v. Thereby, we model time overhead at a charging station for, e. g., parking the car or swapping the battery.

As mentioned above, we represent the bivariate charging function cf_v of a vertex $v \in S$ with a univariate function $\tilde{\mathrm{cf}}_v$, as follows. Consider the *inverse function* cf_v^{-1} mapping a desired departure SoC $b \in [b_v^{\min}, b_v^{\max})$ to the required charging time $x \in \mathbb{R}_{\geq 0}$ when the arrival SoC is 0, i. e., $\mathrm{cf}_v^{-1}(b) = x$ implies that $\tilde{\mathrm{cf}}_v(x) = \mathrm{cf}_v(0, x) = b$. Since $\tilde{\mathrm{cf}}_v$ is strictly increasing on the interval $[b_v^{\min}, b_v^{\max})$ by definition, the function cf_v^{-1} is well-defined on the domain $[b_v^{\min}, b_v^{\max})$. Given the minimum charging time $x_v^{\max} \in \mathbb{R}_{\geq 0}$ required to charge to an SoC $b_v^{\max}$ at v from an arrival SoC of 0, we define the *expanded inverse function* $\tilde{\mathrm{cf}}_v^{-1} \colon [0, M] \to \mathbb{R}_{\geq 0}$ by setting

$$\tilde{\mathrm{cf}}_v^{-1}(b) := \begin{cases} 0 & \text{if } b < b_v^{\min}, \\ x_v^{\max} & \text{if } b \geq b_v^{\max}, \\ \mathrm{cf}_v^{-1}(b) & \text{otherwise.} \end{cases}$$

This yields the equivalence $\mathrm{cf}_v(b, x) = \tilde{\mathrm{cf}}_v(\tilde{\mathrm{cf}}_v^{-1}(b) + x)$ for $b \in [0, M]$ with $b \leq b_v^{\max}$ and $x \in \mathbb{R}_{\geq 0}$; see Figure 5.2. Further, we denote by $\mathrm{cf}_v^{-1}(b_1, b_2) := \tilde{\mathrm{cf}}_v^{-1}(b_2) - \tilde{\mathrm{cf}}_v^{-1}(b_1)$ the time to charge the battery from some arrival SoC $b_1 \in [0, M]$ to a desired departure SoC $b_2 \in [0, M]$ with $b_1 \leq b_2$.

Existing models of charging functions use linear, polynomial, and exponential functions, or piecewise combinations thereof [Mon+17, Pel+17, SDK17]. Typically, these functions are also *concave* with respect to charging time (i. e., charging speed only decreases as the battery's SoC increases). However, charging functions in our model are not limited to such functions per se. Section 5.2.3 discusses necessary conditions for charging functions besides those mentioned above (continuity, monotonicity, and the shifting property) to ensure that our algorithms terminate. For the sake of simplicity and motivated by data input in our experimental evaluation (see Section 5.4.1), examples in subsequent sections use piecewise linear, concave charging functions.

We consider the following objective: For a given source $s \in V$, a target $t \in V$, and an initial SoC $b_s \in [0, M]$, we want to find a feasible s–t path that minimizes overall *trip time*, i. e., the sum of driving time and total time spent at charging stations. Note that if the input graph contains no charging stations ($S = \emptyset$), we have an instance of the basic problem from Section 5.1, hence the considered problem is $\mathcal{NP}$-hard, too.

5.2.2 Basic Approach

Since charging functions are continuous, there is no straightforward way to apply the bicriteria algorithm described in Section 5.1 to our setting: This might require an infinite number of nondominated labels after settling a charging station with a continuous charging function. In this section, we show how the algorithm can be to generalized to our setting. The *charging function propagating (CFP)* algorithm extends labels to maintain infinite, continuous sets of solutions. The core idea is that a label represents all possible tradeoffs between charging time and resulting SoC induced by the last visited charging station (if it exists). First, we show how to represent paths containing charging stations with labels of constant size. Afterwards, we describe the CFP algorithm itself. In the following Section 5.2.3, we discuss the implementation of a crucial part, namely, generating new labels at charging stations.

Labels and SoC Functions. Assume we are given a path P from the source $s \in V$ to some vertex $v \in V$, such that P contains a charging station $u \in S$ and the arrival SoC at u is $b_u \in [0, M]$. Every possible charging time $x_{\text{charge}} \in [0, \text{cf}_u^{-1}(b_u, b_u^{\max})]$ at u results in a certain trip time and an SoC at v. In general, this yields infinitely many feasible, nondominated pairs of trip time and corresponding SoC for the path. We implicitly represent these pairs in one label by storing the charging station u in the label. However, this no longer allows us to apply battery constraints on-the-fly: For vertices visited after u, labels have no fixed SoC, as it depends on how much energy is charged at u. Hence, we compute the SoC profile $f_{[u,\ldots,v]}$ of the subpath from u to v; see Section 4.1.2. The label $\ell = (x_{\text{trip}}, b_u, u, f_{[u,\ldots,v]})$ at the vertex v then consists of the trip time $x_{\text{trip}} \in \mathbb{R}_{\geq 0}$ of the path from s to v (including charging time on every previous charging station except u on the path from s to u), the SoC b_u when reaching u, the last visited charging station u, and the SoC profile $f_{[u,\ldots,v]}$ of the subpath from u to v. Recall that this SoC profile can be represented by three values; see Section 4.1.3. Consequently, even though charging functions can have arbitrary descriptive complexity, we propagate them using labels of constant size. The trip time x_{trip} excludes charging at u, but includes its initialization time $x_{\text{init}}(u)$. Thus, we can think of x_{trip} as the least trip time to reach v if we stop at u (and ignore battery constraints on the u–v path).

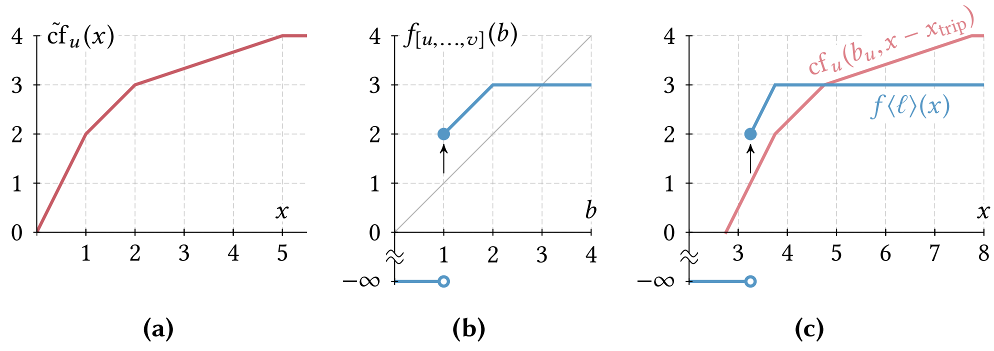


Figure 5.3: Constructing the SoC function $f\langle\ell\rangle$ of a given label $\ell = (x_{\mathrm{trip}}, b_u, u, f_{[u,\dots,v]})$, with $x_{\mathrm{trip}} = 3$ and $b_u = 0.5$. (a) The function $\tilde{\mathrm{cf}}_u$. Assume that the initialization time at u is $x_{\mathrm{init}}(u) = 0$. (b) The SoC profile $f_{[u,\dots,v]}$ of the u–v subpath. Note that the path has negative consumption (the SoC increases as indicated by the arrow). (c) The SoC function $f\langle\ell\rangle$. The function $\mathrm{cf}_u(b_u, x - x_{\mathrm{trip}})$ (red) reflects pairs of trip time and SoC when charging at u, but ignores consumption on the u–v subpath. It is equivalent to the function obtained after shifting $\tilde{\mathrm{cf}}_u$ to the right by $x_{\mathrm{trip}} = 3$ minus $\mathrm{cf}_u^{-1}(0, b_u) = 0.25$. We apply battery constraints with respect to the u–v subpath to this function and obtain the depicted SoC function $f\langle\ell\rangle$ (blue). Its minimum feasible trip time is $x_{\min}(\ell) = 3.25$, because we must spend a charging time of at least 0.25 at u. Moreover, we obtain $f\langle\ell\rangle(x) = 3$ for $x \geq 3.75$ (charging beyond an SoC of 2 at u never pays off, as it wastes energy gains from recuperation).

Accordingly, we define the *SoC function* $f\langle\ell\rangle$ of a label ℓ, to represent all feasible pairs of trip time and SoC associated with the label $\ell = (x_{\mathrm{trip}}, b_u, u, f_{[u,\dots,v]})$. The SoC function $f\langle\ell\rangle\colon \mathbb{R}_{\geq 0} \to [0, M] \cup \{-\infty\}$ mapping trip time to SoC is given as

$$f\langle\ell\rangle(x) := \begin{cases} f_{[u,\dots,v]}(\mathrm{cf}_u(b_u, x - x_{\mathrm{trip}})) & \text{if } x \geq x_{\mathrm{trip}}, \\ -\infty & \text{otherwise.} \end{cases} \tag{5.1}$$

To obtain the value $f\langle\ell\rangle(x)$, i. e., the (arrival) SoC at v for a trip time of $x \geq x_{\mathrm{trip}}$, Equation 5.1 first evaluates the SoC $\mathrm{cf}_u(b_u, x - x_{\mathrm{trip}})$ after charging at u for a total time of $x - x_{\mathrm{trip}}$ with an arrival SoC of b_u. Afterwards, the SoC profile $f_{[u,\dots,v]}$ is applied, which takes account of energy consumption (respecting battery constraints) on the path from the charging station u to the current vertex v. This yields the desired SoC at v. Note that $f\langle\ell\rangle$ can evaluate to $-\infty$ for values greater than x_{trip}, due to battery constraints applied by the SoC profile $f_{[u,\dots,v]}$. We denote by

$$x_{\min}(f\langle\ell\rangle) := \min\{x \in \mathbb{R}_{\geq 0} \mid f\langle\ell\rangle(x) \neq -\infty\}$$

the smallest value for which $f\langle\ell\rangle$ is greater than $-\infty$, i. e., the minimum trip time required for the path represented by ℓ to be feasible. Figure 5.3 shows an example of an SoC function of a label. The definition of SoC functions reflects the interpretation of our labels, which represent all possible tradeoffs between charging time and resulting

SoC on the considered path, induced by the charging function cf_u that belongs to the last charging station u.

Algorithm Description. We are now ready to describe the actual CFP algorithm, which is outlined in Figure 5.4. It propagates labels that are quadruples as defined above. Given two labels ℓ and ℓ', we say that ℓ *dominates* ℓ' if $f\langle\ell\rangle(x) \geq f\langle\ell'\rangle(x)$ holds for all $x \in \mathbb{R}_{\geq 0}$. The *key* of a label ℓ, denoted $\mathrm{key}(\ell) := x_{\min}(f\langle\ell\rangle)$, is defined as its minimum feasible trip time. Note that this value does not have to be stored explicitly in the label, but can be computed on-the-fly by evaluating the inverse of the charging function of the previous station at the minimum SoC for which the subpath from this station to the current vertex becomes feasible.

The algorithm stores two sets $L_{\mathrm{set}}(v)$ and $L_{\mathrm{uns}}(v)$ for each vertex $v \in V$, containing *settled* (i. e., extracted) and *unsettled* labels, respectively. Sets $L_{\mathrm{uns}}(\cdot)$ are organized as priority queues (implemented as binary heaps), allowing efficient extraction of the unsettled label with minimum key (breaking ties by the corresponding SoC of a label). We maintain the invariant that for each $v \in V$, $L_{\mathrm{uns}}(v)$ is empty or the minimum label ℓ (with respect to its key) in $L_{\mathrm{uns}}(v)$ is not dominated by any label in $L_{\mathrm{set}}(v)$. Every time the minimum element of the heap changes, because an element is removed or added, we check whether the new minimum is dominated by a label in $L_{\mathrm{set}}(v)$ and remove it in this case (as it cannot lead to an optimal solution). For piecewise-defined SoC functions, a dominance test requires a linear scan over the subfunctions of both SoC functions. By using heaps for unsettled labels, we avoid unnecessary dominance checks for labels that are never settled. (A more straightforward variant could follow the basic algorithm outlined in Section 5.1 to identify dominated labels, but this lead to slower running times in our tests.)

Given a source $s \in V$, a target $t \in V$, and the initial SoC $b_s \in [0, M]$, the algorithm is initialized in lines 1–8 of Figure 5.4 with a single label $(0, b_s, v^*, \mathrm{id})$ at the source s, while all other label sets are empty. Note that v^* is a special vertex that is (temporarily) added to the graph as a charging station with the charging function $\mathrm{cf}_{v^*} \equiv b_s$. Thereby, we avoid explicit handling of special cases when reaching the first actual charging station. The SoC profile stored in the label is initialized with the identity function (i. e., the SoC is not affected when applying this function). The source vertex is also inserted into a priority queue. The key of a vertex $v \in V$ in the priority queue is the key of the minimum element in $L_{\mathrm{uns}}(v)$, i. e., the minimum feasible trip time among the SoC functions of all unsettled labels.

The algorithm then proceeds along the lines of the BSP algorithm. In each step of the main loop, it first extracts a vertex $v \in V$ with minimum key (breaking ties by SoC) from the priority queue and settles it; see lines 10–12 of Figure 5.4. Note that at this point, the key of the corresponding label $\ell = (x_{\mathrm{trip}}, b_u, u, f_{[u,\dots,v]})$ extracted from $L_{\mathrm{uns}}(v)$ is not greater than that of any label that has not been settled yet.

```
// initialize label sets
foreach $v \in V$ do
    $L_{\text{set}}(v) \longleftarrow \emptyset$
    $L_{\text{uns}}(v) \longleftarrow \emptyset$
$v^* \longleftarrow$ dummy vertex that is (temporarily) added to $V$
$S \longleftarrow S \cup \{v^*\}$
$\tilde{\text{cf}}_{v^*} \longleftarrow [(0, b_s)]$
$L_{\text{uns}}(s) \longleftarrow \{(0, b_s, v^*, \text{id})\}$
$Q$.insert($s$, 0)
// run main loop
while $Q$.isNotEmpty() do
    $v \longleftarrow Q$.minElement()
    $\ell = (x_{\text{trip}}, b_u, u, f_{[u,\dots,v]}) \longleftarrow L_{\text{uns}}(v)$.deleteMin()
    $L_{\text{set}}(v)$.insert($\ell$)
    if $v = t$ then
        return $x_{\min}(f\langle\ell\rangle)$
    // handle charging stations; see Section 5.2.3
    if $v \in S \setminus \{u\}$ then
        foreach $x_{\text{charge}} \in X(\sigma^{\ell}_{\text{old}}, \sigma^{\ell}_{\text{new}}) \setminus \{\infty\}$ do
            $L_{\text{uns}}(v)$.insert(($x_{\text{trip}} + x_{\text{charge}} + x_{\text{init}}(v), f\langle\ell\rangle(x_{\text{trip}} + x_{\text{charge}}), v, \text{id}$))
    // update priority queue
    if $L_{\text{uns}}(v)$.isNotEmpty() then
        $\ell' \longleftarrow L_{\text{uns}}(v)$.minElement()
        $Q$.update($v$, key($\ell'$))
    else
        $Q$.deleteMin()
    // scan outgoing edges
    foreach $(v, w) \in E$ do
        $f_{[u,\dots,w]} \longleftarrow \text{link}(f_{[u,\dots,v]}, f_{(v,w)})$
        if $f_{[u,\dots,w]}(b_u^{max}) \neq -\infty$ then
            $\ell' \longleftarrow (x_{\text{trip}} + d(v,w), b_u, u, f_{[u,\dots,w]})$
            $L_{\text{uns}}(w)$.insert($\ell'$)
            if $\ell' = L_{\text{uns}}(w)$.minElement() then
                $Q$.update($w$, key($\ell'$))
```

Figure 5.4: Pseudocode of the CFP algorithm. It requires an input graph $G = (V, E)$ with cost functions $d\colon E \to \mathbb{R}_{\geq 0}$ and $c\colon E \to \mathbb{R}$, a set $S \subseteq V$ of charging stations, a charging function cf_v for each $v \in S$, a source $s \in V$, a target $t \in V$, a battery capacity $M \in \mathbb{R}_{\geq 0}$, and an initial SoC $b_s \in [0, M]$. The output of the algorithm is the minimum trip time from s to t.

Next, we check whether v is a charging station that differs from the one stored in the current label ℓ, i. e., $v \in S \setminus \{u\}$. If this is the case, we create new labels to incorporate possible recharging at v; see lines 15–17. This means that we have to spawn new labels ℓ' that replace the previous charging station u with v. We can do so by fixing a charging time $x_{\text{charge}} \in \mathbb{R}_{\geq 0}$ at u. For the resulting SoC at u, we evaluate the SoC profile $f_{[u,\dots,v]}$ of the u–v path to determine the SoC at v. We update the trip time accordingly by adding the charging time x_{charge} at u and the initialization time $x_{\text{init}}(v)$ at the new charging station v. We obtain the new label

$$\begin{aligned}\ell' &:= (x_{\text{trip}} + x_{\text{charge}} + x_{\text{init}}(v), f_{[u,\dots,v]}(\text{cf}_u(b_u, x_{\text{charge}})), v, \text{id}) \\ &= (x_{\text{trip}} + x_{\text{charge}} + x_{\text{init}}(v), f\langle\ell\rangle(x_{\text{trip}} + x_{\text{charge}}), v, \text{id}).\end{aligned} \tag{5.2}$$

However, we still face the problem that in general, there are infinitely many possible charging times x_{charge} at the previous charging station u held in ℓ. In Section 5.2.3, we show that for realistic models of charging stations, we only have to consider a small (finite) number of relevant charging times at u when charging at v. Thus, spawning a limited number of new labels, each fixing a certain charging time at u and setting the last charging station to v, is sufficient to represent all nondominated solutions. Note that the original label ℓ is not discarded, to reflect the possibility of not stopping at the charging station v.

In lines 18–22 of Figure 5.4, the key of v in the priority queue is updated. Since the label ℓ was settled and new labels may have spawned in case v is a charging station, we update the key of v to the new smallest key of an unsettled label, if it exists. Otherwise, v is removed from the queue.

Afterwards, we scan all outgoing edges $(v, w) \in E$; see lines 23–29. Given the current label $\ell = (x_{\text{trip}}, b_u, u, f_{[u,\dots,v]})$, traversing the edge (v, w) means to increase trip time by $d(v, w)$ and apply the (constant-time) link operation to the SoC profile $f_{[u,\dots,v]}$ of ℓ and the SoC profile $f_{(v,w)}$ induced by the energy consumption $c(v, w)$; see Section 4.1.3. We compute $f_{[u,\dots,w]} := \text{link}(f_{[u,\dots,v]}, f_{(v,w)})$ and construct the label

$$\ell' := (x_{\text{trip}} + d(v, w), b_u, u, f_{[u,\dots,w]}).$$

Unless the SoC profile $f_{[u,\dots,w]}$ of ℓ' indicates that the u–w subpath is infeasible, the new label ℓ' is added to the label set at the vertex w. Note that we perform no dominance checks at this point (unless the minimum element in the label set $L_{\text{uns}}(w)$ changes), exploiting the fact that labels are organized in two sets per vertex.

When extracting a label ℓ at the target vertex t for the first time, we pick the least charging time at the last station such that t can be reached, i. e., the minimum feasible trip time $x_{\min}(f\langle\ell\rangle)$ of $f\langle\ell\rangle$, and the algorithm terminates; see line 14 in Figure 5.4. Correctness of the CFP algorithm follows from Lemma 5.1 shown in Section 5.2.3 below and the fact that the first extracted label ℓ at t minimizes the feasible trip time (recall that the algorithm is label setting and minimum feasible trip time is used as key in the priority queue). Theorem 5.2 at the end of Section 5.2.3 summarizes these insights. The asymptotic running time of the algorithm is exponential in the input graph in the worst case (for reasonable charging models; see Section 5.2.3).

For path unpacking, we add two pointers to each label, storing its parent vertex and parent label. For a charging station $v \in S$, the vertex v can be its own parent. Two consecutive identical parents then imply the use of a (previous) charging station $u \in S$, which is stored in the former label. The according charging time is the difference between the trip times of both labels.

5.2.3 Spawning Labels at Charging Stations

As mentioned in Section 5.2.2, the CFP algorithm constructs new labels at charging stations to represent all nondominated solutions. We now prove that for reasonable models of charging functions, it suffices to spawn a small number of labels that replace the previous charging station with the new one. The key idea is that we only require labels that correspond to charging at the station that offers the better charging speed at a certain (relative) point in time. We define *switching sequences* for pairs of functions, containing points at which the charging speed of the new function surpasses the old one. Lemma 5.1 proves that spawning one label per element of the switching sequence suffices. Moreover, switching sequences are finite (and linear in the descriptive complexity) for typical models of charging functions, which implies that the CFP algorithm terminates. Before proving Lemma 5.1, we introduce helpful tools. We also formalize switching sequences and the slope of an SoC function.

Consider a label $\ell = (x_{\text{trip}}, b_u, u, f_{[u,\dots,v]})$ extracted at some charging station $v \in S$. We want to create new labels that reflect charging at v. This requires us to fix a charging time $x_{\text{charge}} \in \mathbb{R}_{\geq 0}$ at the previous station u, so that we can set v as the last visited charging station of a new label ℓ'; see Equation 5.2 and Figure 5.5. We denote the resulting label by $(x_{\text{charge}} \rightarrow \ell) := \ell'$, as it is obtained after setting the charging time in ℓ to x_{charge}. Recall that in the label $x_{\text{charge}} \rightarrow \ell$, we replace the old charging station u with the new station v. Moreover, we set $x_{\text{trip}} + x_{\text{charge}} + x_{\text{init}}(v)$ as its overall trip time and $f\langle \ell \rangle(x_{\text{trip}} + x_{\text{charge}})$ as the corresponding arrival SoC at v. The SoC function $f\langle x_{\text{charge}} \rightarrow \ell \rangle$ represents all tradeoffs between charging time at the new charging station v and resulting SoC. If the label $x_{\text{charge}} \rightarrow \ell$ is not feasible, i. e., $x_{\text{trip}} + x_{\text{charge}} < x_{\min}(f\langle \ell \rangle)$, we obtain $f\langle x_{\text{charge}} \rightarrow \ell \rangle \equiv -\infty$.

Not every charging time $x_{\text{charge}} \in \mathbb{R}_{\geq 0}$ at u yields a reasonable solution. In particular, if we can find a charging time $x'_{\text{charge}} \in \mathbb{R}_{\geq 0}$ such that $f\langle x'_{\text{charge}} \rightarrow \ell \rangle$ dominates $f\langle x_{\text{charge}} \rightarrow \ell \rangle$, we know that a charging time of x_{charge} is never beneficial. Intuitively, if the new charging station v allows fast charging, it could pay off to charge less energy at the previous station u and spend more time at v instead, so $f\langle x_{\text{charge}} - \varepsilon \rightarrow \ell \rangle$ dominates $f\langle x_{\text{charge}} \rightarrow \ell \rangle$ for some $\varepsilon > 0$. Similarly, if the charging station u offers better charging speed, a charging time $x_{\text{charge}} + \varepsilon$ might be the better

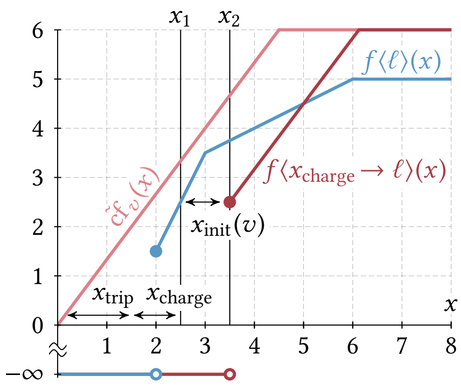


Figure 5.5: Spawning a label at a charging station. Given the SoC function $f\langle \ell \rangle$ of a label $\ell = (x_{\text{trip}}, b_u, u, f_{[u,\dots,v]})$ at a charging station $v \in S$, we can spawn a new label $x_{\text{charge}} \to \ell$ by picking a charging time x_{charge} at the station u. We compare the slopes of $f\langle \ell \rangle$ at $x_1 = x_{\text{trip}} + x_{\text{charge}}$ and $f\langle x_{\text{charge}} \to \ell \rangle$ at $x_2 = x_{\text{trip}} + x_{\text{charge}} + x_{\text{init}}(v)$ to determine which one represents the better choice. Note that x_{trip} is smaller than $x_{\min}(f\langle \ell \rangle)$, due to battery constraints.

choice. In other words, the best choice of the value x_{charge} depends on the slopes of the two SoC functions $f\langle \ell \rangle$ and $f\langle x_{\text{charge}} \to \ell \rangle$.

Recall that we define the *slope* of a given function f at some $x' \in \mathbb{R}_{\geq 0}$ as the corresponding *right derivative* $(\partial f(x)/\partial x)(x')$ to ensure that the slope is well-defined also for piecewise-defined SoC functions and at the minimum feasible trip time of an SoC function. As before, let $\ell = (x_{\text{trip}}, b_u, u, f_{[u,\dots,v]})$ be a label at a charging station $v \in S$. We introduce a function $\sigma^{\ell}_{\text{old}} \colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ that describes the slope of the SoC function $f\langle \ell \rangle$ at $x_{\text{trip}} + x_{\text{charge}}$ as a function of the charging time $x_{\text{charge}} \in \mathbb{R}_{\geq 0}$. Formally, we define

$$\sigma^{\ell}_{\text{old}}(x_{\text{charge}}) := \begin{cases} \frac{\partial f\langle \ell \rangle(x)}{\partial x}(x_{\text{trip}} + x_{\text{charge}}) & \text{if } x_{\text{trip}} + x_{\text{charge}} \geq x_{\min}(f\langle \ell \rangle), \\ \infty & \text{otherwise.} \end{cases}$$

Note that the slope $\sigma^{\ell}_{\text{old}}(x_{\text{charge}})$ of the SoC function $f\langle \ell \rangle$ at $x_{\text{trip}} + x_{\text{charge}}$ is equivalent to the slope of the charging function cf_u of the vertex u for the arrival SoC b_u and the charging time x_{charge}. Hence, we obtain $\sigma^{\ell}_{\text{old}}(x_{\text{charge}}) = (\partial\, \text{cf}_u(b_u, x)/\partial x)(x_{\text{charge}})$ for all $x_{\text{charge}} \in \mathbb{R}_{\geq 0}$, unless battery constraints on the u–v path render a charging time of x_{charge} infeasible or unprofitable (in which case the slope $\sigma^{\ell}_{\text{old}}$ is either ∞ or 0). Thus, $\sigma^{\ell}_{\text{old}}(x_{\text{charge}})$ can be interpreted as the charging speed when continuing to charge the battery at u after a charging time of x_{charge}.

Alternatively, one could interrupt charging at u after a charging time of x_{charge}, continue the journey, and switch to the new charging station upon arrival at v. The charging speed that can be achieved in this case is given by the slope of the SoC function $f\langle x_{\text{charge}} \to \ell \rangle(x)$ at $x = x_{\text{trip}} + x_{\text{charge}} + x_{\text{init}}(v)$; see Figure 5.5. Similar to $\sigma^{\ell}_{\text{old}}$, we define the function $\sigma^{\ell}_{\text{new}} \colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ as

$$\sigma^{\ell}_{\text{new}}(x_{\text{charge}}) := \begin{cases} \frac{\partial f\langle x_{\text{charge}} \to \ell \rangle(x)}{\partial x}(x_{\text{trip}} + x_{\text{charge}} + x_{\text{init}}(v)) & \text{if } x_{\text{trip}} + x_{\text{charge}} \geq x_{\min}(f\langle \ell \rangle), \\ \infty & \text{otherwise.} \end{cases}$$

It maps total charging time x_{charge} at u to the slope of the SoC function $f\langle x_{\text{charge}} \rightarrow \ell\rangle$ at the time $x_{\text{trip}} + x_{\text{charge}} + x_{\text{init}}(v)$ that corresponds to arrival at v and starting to charge the battery. Hence, the value of $\sigma^{\ell}_{\text{new}}(x_{\text{charge}})$ is equivalent to the slope of the new charging function cf_v of v for the arrival SoC $f\langle\ell\rangle(x_{\text{trip}} + x_{\text{charge}})$ and the charging time 0. Formally, we have $\sigma^{\ell}_{\text{new}}(x_{\text{charge}}) = (\partial\,\text{cf}_v(f\langle\ell\rangle(x_{\text{trip}} + x_{\text{charge}}), x)/\partial x)(0)$ if $x_{\text{trip}} + x_{\text{charge}}$ is at least $x_{\min}(f\langle\ell\rangle)$ and $\sigma^{\ell}_{\text{new}}(x_{\text{charge}}) = \infty$ otherwise.

Given the two functions $\sigma^{\ell}_{\text{old}}$ and $\sigma^{\ell}_{\text{new}}$ defined above, we are interested in points in time where $\sigma^{\ell}_{\text{new}}$ surpasses $\sigma^{\ell}_{\text{old}}$, because at such points it may pay off to interrupt charging at u to benefit from a better charging rate at v later on. Additionally, the minimum charging time $x_{\min}(f\langle\ell\rangle) - x_{\text{trip}}$ necessary to reach the new charging station v may be relevant in cases where the charging function of v has low slope but a large minimum SoC value (e. g., if v is a swapping station). We define the *switching sequence* of $\sigma^{\ell}_{\text{old}}$ and $\sigma^{\ell}_{\text{new}}$, denoted

$$X(\sigma^{\ell}_{\text{old}}, \sigma^{\ell}_{\text{new}}) := [x_{\min}(f\langle\ell\rangle) - x_{\text{trip}} = x_1, x_2, \dots, x_{k-1}, x_k = \infty],$$

as the sequence of all candidate points in time to interrupt charging at the old station u, arranged in ascending order. (The value $x_k = \infty$ is only included to avoid additional case distinctions in the proof of Lemma 5.1 below.) Formally, we demand for $X(\sigma^{\ell}_{\text{old}}, \sigma^{\ell}_{\text{new}})$ that $x_1 = x_{\min}(f\langle\ell\rangle) - x_{\text{trip}}$, $x_k = \infty$, $x_i < x_{i+1}$ for $i \in \{1, \dots, k-1\}$, and for all $i \in \{2, \dots, k-1\}$ there exists a value $\varepsilon > 0$ such that for all $0 < \delta < \varepsilon$ it holds that $\sigma^{\ell}_{\text{old}}(x_i - \delta) \geq \sigma^{\ell}_{\text{new}}(x_i - \delta)$ and $\sigma^{\ell}_{\text{old}}(x_i + \delta) < \sigma^{\ell}_{\text{new}}(x_i + \delta)$. Moreover, we assume the sequence $X(\sigma^{\ell}_{\text{old}}, \sigma^{\ell}_{\text{new}})$ to be *maximal*, i. e., it contains all values with the above property. In general, pairs of functions do not necessarily have a switching sequence of finite length $k \in \mathbb{N}$. At the end of this section, we argue that the length of a switching sequence is linear in the descriptive complexity (and thus, finite) for reasonable charging models.

We now prove Lemma 5.1, stating that we only have to spawn a bounded number of new labels at charging stations. In particular, it suffices to add at most one label per element in the switching sequence induced by the last visited charging station in the current label and the new station.

Lemma 5.1. *For a vertex $v \in S$ and a label $\ell = (x_{\text{trip}}, b_u, u, f_{[u,\dots,v]})$ at v, let the (finite) switching sequence of $\sigma^{\ell}_{\text{old}}$ and $\sigma^{\ell}_{\text{new}}$ be given as $X(\sigma^{\ell}_{\text{old}}, \sigma^{\ell}_{\text{new}}) = [x_1, x_2, \dots, x_{k-1}, x_k]$. For every charging time $x_{\text{charge}} \in \mathbb{R}_{\geq 0}$, there exists an $i \in \{1, \dots, k-1\}$ such that the SoC functions $f\langle\ell\rangle$ and $f\langle x_i \rightarrow \ell\rangle$ together dominate the SoC function $f\langle x_{\text{charge}} \rightarrow \ell\rangle$, i. e., for all $x \in \mathbb{R}_{\geq 0}$ we have* $\max\{f\langle\ell\rangle(x), f\langle x_i \rightarrow \ell\rangle(x)\} \geq f\langle x_{\text{charge}} \rightarrow \ell\rangle(x)$.

Proof. Consider an arbitrary charging time $x_{\text{charge}} \in \mathbb{R}_{\geq 0}$ at the previous charging station u and the induced label $x_{\text{charge}} \rightarrow \ell$. If $x_{\text{trip}} + x_{\text{charge}} < x_{\min}(f\langle\ell\rangle)$, the SoC function $f\langle x_{\text{charge}} \rightarrow \ell\rangle \equiv -\infty$ is dominated by $f\langle\ell\rangle$. Hence, we assume in what follows that $x_{\text{charge}} \geq x_1 = x_{\min}(f\langle\ell\rangle) - x_{\text{trip}}$. Then there exists a unique index $i \in \{1, \dots, k-1\}$,

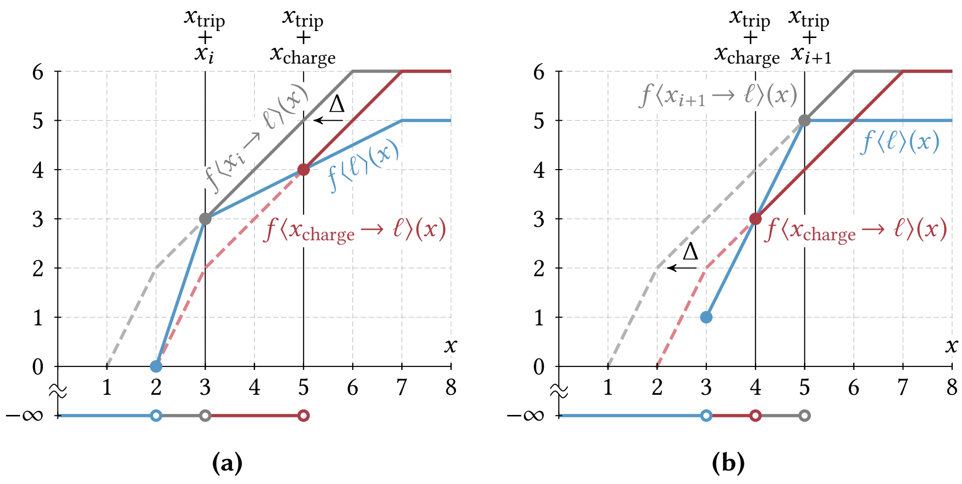


Figure 5.6: Illustration of dominated SoC functions at a charging station $v \in S$. For simplicity, we assume that $x_{\text{init}}(v) = 0$. Dashed segments indicate the (shifted) charging function $\tilde{\text{cf}}_v$ of v. Note that a function dominates the area beneath it. (a) The slope of the SoC function $f\langle \ell\rangle$ is lower than the slope of $f\langle x_{\text{charge}} \to \ell\rangle$ at $x_{\text{trip}} + x_{\text{charge}} = 5$. Hence, it pays off to decrease charging time at the previous station to $x_i = 3$ and charge more energy at v instead. (b) The slope of $f\langle \ell\rangle$ is greater than the slope of $f\langle x_{\text{charge}} \to \ell\rangle$ for $x_{\text{trip}} + x_{\text{charge}} = 4$. Therefore, charging more energy at the previous charging station pays off and the SoC functions $f\langle \ell\rangle$ and $f\langle x_{i+1} \to \ell\rangle$ together dominate the function $f\langle x_{\text{charge}} \to \ell\rangle$.

such that $x_i \le x_{\text{charge}} < x_{i+1}$ holds. We distinguish two cases, depending on the slopes σ^ℓ_{old} and σ^ℓ_{new} at x_{charge} and show that, together with $f\langle \ell\rangle$, the function $f\langle x_i \to \ell\rangle$ or the function $f\langle x_{i+1} \to \ell\rangle$ dominates $f\langle x_{\text{charge}} \to \ell\rangle$.

Case 1: $\sigma^\ell_{\text{old}}(x_{\text{charge}}) < \sigma^\ell_{\text{new}}(x_{\text{charge}})$. Intuitively, this means that the new charging station v provides a better charging speed than the old station u for the considered charging time x_{charge}. Hence, leaving u earlier to charge more energy at v and benefit from faster charging pays off. Consequently, $f\langle x_i \to \ell\rangle$ dominates $f\langle x_{\text{charge}} \to \ell\rangle$, since the charging speed of v is better (or equally good) for all $x \in [x_i, x_{\text{charge}}]$; see Figure 5.6a for an example. Since $x_i \le x_{\text{charge}}$ and a charging time of x_i is sufficient to reach v, we have $x_{\min}(f\langle x_i \to \ell\rangle) \le x_{\min}(f\langle x_{\text{charge}} \to \ell\rangle)$. In other words, if the SoC function induced by x_{charge} is finite for some $x \in \mathbb{R}_{\ge 0}$, so is the function induced by x_i. Further, recall that the vertex v is reached with an arrival SoC of $f\langle \ell\rangle(x_{\text{trip}} + x_i)$ or $f\langle \ell\rangle(x_{\text{trip}} + x_{\text{charge}})$ when the charging time at u is set to x_i or x_{charge}, respectively. By assumption, the slope $\sigma^\ell_{\text{new}}(x_{\text{charge}})$ is strictly positive, which means that energy can still be charged at v after the SoC has reached $f\langle \ell\rangle(x_{\text{trip}} + x_{\text{charge}})$, so it must hold that $f\langle \ell\rangle(x_{\text{trip}} + x_{\text{charge}}) < b^{\max}_v$. Hence, we can define the value

$$\Delta_{\text{charge}} := \text{cf}^{-1}_v(f\langle \ell\rangle(x_{\text{trip}} + x_i), f\langle \ell\rangle(x_{\text{trip}} + x_{\text{charge}}))$$

that corresponds to the time to recharge the gap in arrival SoC between $f\langle x_i \rightarrow \ell\rangle$ and $f\langle x_{\mathrm{charge}} \rightarrow \ell\rangle$ at the new station v. For arbitrary values $x \geq x_{\min}(f\langle x_{\mathrm{charge}} \rightarrow \ell\rangle)$, we exploit the shifting property to obtain

$$\begin{aligned} f\langle x_i \rightarrow \ell\rangle(x) &= \mathrm{cf}_v(f\langle\ell\rangle(x_{\mathrm{trip}} + x_i), x - x_i - x_{\mathrm{trip}} - x_{\mathrm{init}}(v)) \\ &= \mathrm{cf}_v(f\langle\ell\rangle(x_{\mathrm{trip}} + x_{\mathrm{charge}}), x - x_i - x_{\mathrm{trip}} - x_{\mathrm{init}}(v) - \Delta_{\mathrm{charge}}) \\ &= f\langle x_{\mathrm{charge}} \rightarrow \ell\rangle(x + x_{\mathrm{charge}} - x_i - \Delta_{\mathrm{charge}}). \end{aligned}$$

We claim that $\Delta_{\mathrm{trip}} := x_{\mathrm{charge}} - x_i - \Delta_{\mathrm{charge}} \geq 0$ holds. This follows immediately from the fact that $\sigma^{\ell}_{\mathrm{new}}(x) \geq \sigma^{\ell}_{\mathrm{old}}(x)$ holds for all $x \in [x_i, x_{\mathrm{charge}}]$. Thus, the time Δ_{charge} spent at v cannot take longer than $x_{\mathrm{charge}} - x_i$, the time to charge the same amount of energy at u. Consequently, we obtain $\Delta_{\mathrm{trip}} \geq 0$. This, in turn, implies that for all $x \geq x_{\min}(f\langle x_{\mathrm{charge}} \rightarrow \ell\rangle)$, we have

$$\begin{aligned} f\langle x_i \rightarrow \ell\rangle(x) &= f\langle x_{\mathrm{charge}} \rightarrow \ell\rangle(x + \Delta_{\mathrm{trip}}) \\ &\geq f\langle x_{\mathrm{charge}} \rightarrow \ell\rangle(x). \end{aligned}$$

Case 2: $\sigma^{\ell}_{\mathrm{old}}(x_{\mathrm{charge}}) \geq \sigma^{\ell}_{\mathrm{new}}(x_{\mathrm{charge}})$. In this case, the charging station u offers a more (or equally) favorable charging speed, so it pays off to spend more time charging at u. Recall that $f\langle\ell\rangle(x_{\mathrm{trip}} + x_{\mathrm{charge}}) = f\langle x_{\mathrm{charge}} \rightarrow \ell\rangle(x_{\mathrm{trip}} + x_{\mathrm{charge}} + x_{\mathrm{init}}(v))$ holds by definition. Given that the slope $\sigma^{\ell}_{\mathrm{old}}$ of $f\langle\ell\rangle$ is greater or equal for all $x \in [x_{\mathrm{charge}}, x_{i+1})$, it follows that $f\langle\ell\rangle(x_{\mathrm{trip}} + x) \geq f\langle x_{\mathrm{charge}} \rightarrow \ell\rangle(x_{\mathrm{trip}} + x + x_{\mathrm{init}}(v))$ holds for arbitrary values $x \in [x_{\mathrm{charge}}, x_{i+1})$. For real-valued $x \geq x_{i+1}$ (which only exist if $i + 1 < k$), we proceed along the lines of the first case to obtain a nonnegative value $\Delta_{\mathrm{trip}} \geq 0$ that equals the difference between the time to charge from $f\langle\ell\rangle(x_{\mathrm{trip}} + x_{\mathrm{charge}})$ to $f\langle\ell\rangle(x_{\mathrm{trip}} + x_{i+1})$ at u and v, respectively. Since u offers a charging speed at least as high as the one at v in the whole interval $[x_{\mathrm{charge}}, x_{i+1}]$, this difference, and therefore Δ_{trip}, is again nonnegative. Thus, we can show (similar to Case 1) that

$$\begin{aligned} f\langle x_{i+1} \rightarrow \ell\rangle(x) &= f\langle x_{\mathrm{charge}} \rightarrow \ell\rangle(x + \Delta_{\mathrm{trip}}) \\ &\geq f\langle x_{\mathrm{charge}} \rightarrow \ell\rangle(x) \end{aligned}$$

holds for arbitrary real-valued $x \geq x_{i+1}$; see Figure 5.6b for an illustration. Altogether, we obtain that $\max\{f\langle\ell\rangle(x), f\langle x_{i+1} \rightarrow \ell\rangle(x)\} \geq f\langle x_{\mathrm{charge}} \rightarrow \ell\rangle(x)$ holds for $x \in \mathbb{R}_{\geq 0}$, which completes the proof of the second case. ☐

Given the label $\ell = (x_{\mathrm{trip}}, b_u, u, f_{[u,\ldots,v]})$ at the charging station $v \in S$, we spawn one new label $x \rightarrow \ell$ for each (finite) element $x \in X(\sigma^{\ell}_{\mathrm{old}}, \sigma^{\ell}_{\mathrm{new}})$ in the switching sequence. If the label is not dominated, it is added to the label set at v; see lines 16–17 in Figure 5.4. For instance, in the special case that u or v is a swapping station, the switching sequence $[x_{\min}(f\langle\ell\rangle) - x_{\mathrm{trip}}, \infty]$ consists of two values and exactly one new label is spawned at v (which corresponds to spending the minimum amount of charging time at u such that v can be reached). Recall that the original label ℓ is not thrown away, reflecting the possibility of not using the charging station v. Furthermore, Lemma 5.1 implies that all nondominated solutions that extend ℓ by charging at v are computed by the algorithm. By induction, correctness is also maintained for routes with two or more charging stops. Therefore, we obtain Theorem 5.2 given below.

Theorem 5.2. *If the switching sequences induced by arbitrary pairs of charging functions have finite length, the CFP algorithm terminates and finds the shortest feasible path between a given pair of vertices $s \in V$ and $t \in V$ for a given initial SoC $b_s \in [0, M]$.*

Computing Switching Sequences. In a practical implementation of the CFP algorithm, we need to be able to efficiently compute the switching sequences for given labels and charging functions; c. f. line 16 in Figure 5.4. If certain properties of the available charging functions are known, a reasonable approach is to derive the switching sequences for such functions analytically beforehand and provide a specialized implementation for them.

To give an example, we discuss the case where all charging functions in the network (and hence, all SoC functions) are *piecewise linear* and *concave*, as is the case in our experimental study (see Section 5.4.1). We show how a superset of the switching sequence is easily determined in this case. Given a label $\ell = (x_{\text{trip}}, b_u, u, f_{[u,\dots,v]})$ at a charging station $v \in S$, let its piecewise linear SoC function $f\langle\ell\rangle$ be given as a sequence $F = [(x_1, b_1), \dots, (x_k, b_k)]$ of breakpoints. In other words, we have $f\langle\ell\rangle(x) = -\infty$ for $x < x_1$, $f\langle\ell\rangle(x) = b_k$ for $x \geq x_k$, and for values $x_i \leq x < x_{i+1}$, with $i \in \{1, \dots, k-1\}$, we evaluate the function by linear interpolation between the breakpoints (x_i, b_i) and (x_{i+1}, b_{i+1}); see Section 3.1 for details. Observe that we have $x_1 = x_{\min}(f\langle\ell\rangle)$. The following Lemma 5.3 shows that the switching sequence of the slopes induced by $f\langle\ell\rangle$ and the charging function cf_v of v must be a subsequence of $[x_1 - x_{\text{trip}}, \dots, x_k - x_{\text{trip}}, \infty]$. Note that in particular, the switching sequence does not depend on the charging function cf_v at v.

Lemma 5.3. *Given a label ℓ at a vertex $v \in S$, let its piecewise linear and concave SoC function $f\langle\ell\rangle$ be defined by the sequence $F = [(x_1, b_1), \dots, (x_k, b_k)]$ of breakpoints. Similarly, let the charging function $\tilde{\text{cf}}_v$ of v be piecewise linear and concave. The switching sequence $X(\sigma^\ell_{\text{old}}, \sigma^\ell_{\text{new}})$ is a subsequence of $\bar{X}(\sigma^\ell_{\text{old}}, \sigma^\ell_{\text{new}}) := [x_1 - x_{\text{trip}}, \dots, x_k - x_{\text{trip}}, \infty]$, i. e., $X(\sigma^\ell_{\text{old}}, \sigma^\ell_{\text{new}})$ contains only values that are also contained in $\bar{X}(\sigma^\ell_{\text{old}}, \sigma^\ell_{\text{new}})$.*

Proof. Since both $f\langle\ell\rangle$ and $\tilde{\text{cf}}_v$ are piecewise linear, their corresponding slope functions σ^ℓ_{old} and σ^ℓ_{new} are *piecewise constant* functions, namely, $\sigma^\ell_{\text{old}} \equiv (b_{i+1} - b_i)/(x_{i+1} - x_i)$ holds in the interval $[x_i - x_{\text{trip}}, x_{i+1} - x_{\text{trip}})$ for arbitrary $i \in \{1, \dots, k-1\}$, and an analogous statement holds for σ^ℓ_{new}. This implies that each value of the switching sequence must correspond to a breakpoint of σ^ℓ_{old} or σ^ℓ_{new}, because both functions are constant between these breakpoints. Moreover, since $f\langle\ell\rangle$ and $\tilde{\text{cf}}_v$ are both concave on their subdomain with finite image, the functions σ^ℓ_{old} and σ^ℓ_{new} are *decreasing*. Consequently, the slope σ^ℓ_{new} can surpass σ^ℓ_{old} only at the breakpoints of σ^ℓ_{old}, i. e., at points where the latter function decreases. The x-coordinates of these breakpoints are exactly the finite values of $\bar{X}(\sigma^\ell_{\text{old}}, \sigma^\ell_{\text{new}})$. □

Lemma 5.3 implies that simply spawning one new label for each breakpoint of $f\langle\ell\rangle$ is sufficient to maintain correctness. Unnecessary labels are detected implicitly during dominance checks. Considering the more general case where charging functions are piecewise linear (but not necessarily concave), it is not hard to see that the length of the switching sequence must be linear in the number of breakpoints of both considered functions. Similar observations can be made for other realistic models of charging functions based on, e. g., exponential functions or piecewise combinations of linear and exponential functions.

5.2.4 A* Search

To accelerate the CFP algorithm, we present techniques that extend A* search [HNR68, MP10]. The basic idea of A* search is to use vertex potentials that guide the search towards the target, in order to reduce the search space. The potential of a vertex is added to the key of a label when updating the priority queue in line 20 or line 29 of the algorithm in Figure 5.4. Thereby, vertices that are closer to the target get smaller keys. Below, we first generalize the notion of potential consistency to our setting by incorporating the SoC at a vertex, before we introduce different consistent potential functions. They attempt to improve known potential functions by estimating the remaining charging time that is required to reach the target.

Consistency of Potentials. We aim at potential functions that are based on backward searches from the target vertex $t \in V$, providing lower bounds on the trip time from any vertex to t. A consistent potential function is easily obtained from a single-criterion backward search as follows. It runs Dijkstra's algorithm [Dij59] on the backward graph $\bar{G}$ from t, using the cost function d, which represents driving time on edges. This yields, for each vertex $v \in V$, its minimum (unconstrained) driving time to reach t. These lower bounds on the remaining trip time induce a consistent potential function on the vertices [DMS08, TC92].

We can do better, by exploiting that the trip time from v to t depends on the SoC of a label. (Observe that both the charging time as well as the route and hence, the driving time, of an optimal solution can change for different SoC values.) We propose a potential function $\pi\colon V \times [0,M] \cup \{-\infty\} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ taking the current SoC into account, such that $\pi(v,b)$ yields a lower bound on the trip time from v to t if the SoC at v is $b \in [0,M] \cup \{-\infty\}$. We define $\pi(v,-\infty) := \infty$. We now discuss how correctness of our approach can be maintained in the presence of a potential function that incorporates the SoC.

First, we generalize the notion of consistency of a potential function. We say that a potential function $\pi\colon V \times [0,M] \cup \{-\infty\} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ is *consistent* if it results in nonnegative reduced driving times, i. e., we get

$$\underline{d}((u,v),b) := d(u,v) - \pi(u,b) + \pi(v, f_{(u,v)}(b)) \geq 0$$

for all edges $(u,v) \in E$ and values $b \in [0,M]$, where $f_{(u,v)}$ is the SoC profile of (u,v). Second, for the SoC function $f\langle\ell\rangle$ representing a label $\ell = (x_{\mathrm{trip}}, b_u, u, f_{[u,\dots,v]})$ at a vertex $v \in V$, we define its *consistent key* (to be used in the priority queue of CFP) as $\mathrm{key}^*(\ell) := \min_{x \in \mathbb{R}_{\geq 0}} x + \pi(v, f\langle\ell\rangle(x))$. We claim that consecutive consistent keys of labels generated after edge scans are increasing if the potential function π is consistent. To see this, consider a label ℓ at a vertex $u \in V$ and the label ℓ' that is created after scanning an edge $(u,v) \in E$. Recall that $f\langle\ell'\rangle(x + d(u,v)) = f_{(u,v)}(f\langle\ell\rangle(x))$ holds by construction of ℓ', so we can substitute $x' := x + d(u,v)$ below to get

$$\begin{aligned}
\mathrm{key}^*(\ell) &= \min_{x \in \mathbb{R}_{\geq 0}} x + \pi(u, f\langle\ell\rangle(x)) \\
&\leq \min_{x \in \mathbb{R}_{\geq 0}} x + d(u,v) + \pi(v, f_{(u,v)}(f\langle\ell\rangle(x))) \\
&= \min_{x \in \mathbb{R}_{\geq 0}} x + \pi(v, f\langle\ell'\rangle(x)) \\
&= \mathrm{key}^*(\ell').
\end{aligned}$$

Similarly, we have to ensure that labels spawned at charging stations never have a smaller consistent key than the original label. Consider the charging function cf_v of a vertex $v \in S$. We denote by $\mathrm{cf}_{\max}(v)$ the *maximum slope* of the charging function cf_v, which typically equals $\mathrm{cf}^*_{\max}(v) := \max_{x \in \mathbb{R}_{>0}} \partial\, \tilde{\mathrm{cf}}_v(x)/\partial x$. (As before, we use the *right* derivative to ensure that slope is well-defined for piecewise-defined functions.) In the special case that $\tilde{\mathrm{cf}}_v(0) \neq 0$, we incorporate initialization time to obtain a finite slope $b_v^{\min}/x_{\mathrm{init}}(v)$ for the initial SoC gain, presuming that $x_{\mathrm{init}}(v) \neq 0$. In total, we obtain the maximum slope $\mathrm{cf}_{\max}(v) := \max\{\mathrm{cf}^*_{\max}(v), b_v^{\min}/x_{\mathrm{init}}(v)\}$ of v. For instance, we get $\mathrm{cf}_{\max}(v) = M/x_{\mathrm{init}}(v)$ if v is a swapping station. To ensure that labels spawned at the vertex v do not result in decreasing keys, we demand for the slope of the potential $\pi(v,b)$ at v that

$$\frac{\partial \pi(v,b)}{\partial b} \geq -\frac{1}{\mathrm{cf}_{\max}(v)}.$$

We say that the potential π *overestimates charging speed* at v in this case. Observe that overestimation of charging speed implies that the term $x + \pi(v, \tilde{\mathrm{cf}}(x))$ is *increasing* for $x \in \mathbb{R}_{\geq 0}$, so charging at v does not decrease the potential.

Finally, we demand that the potential at the target is $\pi(t,b) = 0$ for arbitrary SoC $b \in [0,M]$. In summary, when using consistent keys, there are the following three requirements to a potential function π.

1. The potential function π is consistent.
2. Potentials at charging stations overestimate charging speed.
3. The target vertex has a potential of 0 (for any finite SoC).

Then, the algorithm is label setting and the correct result is obtained as soon as t is reached (since the key at t equals trip time, so any label extracted at a later point must have a higher trip time). As a simple example, consider the plain CFP algorithm described in Section 5.2.2, which uses no potential function. This is equivalent to a potential function that evaluates to 0 at all vertices for arbitrary SoC. Clearly, the smallest feasible trip time of $f\langle\ell\rangle$ is in fact the consistent key of a label ℓ. Moreover, observe that the potential function $\pi \equiv 0$ is consistent, overestimates charging speed, and equals 0 at the target. In what follows, we derive more sophisticated potential functions, which make the search goal directed. For each potential function, we show that the requirements listed above are fulfilled.

Potentials Based on Single-Criterion Search. We introduce our first consistent potential function. To obtain a lower bound on the necessary charging time on the path from some vertex $v \in V$ to the target, let $\mathrm{cf}_{\max} := \max_{v \in S} \mathrm{cf}_{\max}(v)$ denote the maximum slope of *any* charging function in S, i. e., the maximum charging speed available in the network. We define a new cost function $\omega\colon E \to \mathbb{R}$, which is given as $\omega(e) := d(e) + (c(e)/\,\mathrm{cf}_{\max})$ for an edge $e \in E$. This function adds to the driving time of every edge a lower bound on the time that is required for charging the energy consumed along the edge. Note that the bound can become negative for some edges, due to negative consumption values.

Given the target vertex $t \in V$, prior to running CFP, we perform three (single-criterion) runs of Dijkstra's algorithm from t on the backward graph $\bar{G}$, each using one of the cost functions d, c, and ω, respectively. Thereby, we obtain, for every vertex $v \in V$, the distances $\mathrm{dist}_d(v,t)$, $\mathrm{dist}_c(v,t)$, and $\mathrm{dist}_\omega(v,t)$ from v to t with respect to the cost functions d, c, and ω. Note that the computation of $\mathrm{dist}_c(v,t)$ and $\mathrm{dist}_\omega(v,t)$ is label correcting, due to negative costs. We can apply potential shifting [Joh77] to remedy this issue (c. f. Section 4.2.1), but the effect on overall running time is negligible in practice. Using the obtained distances, the potential function $\pi_\omega\colon V \times [0,M] \cup \{-\infty\} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ is defined by

$$\pi_\omega(v,b) := \begin{cases} \mathrm{dist}_d(v,t) & \text{if } b \geq \mathrm{dist}_c(v,t), \\ \mathrm{dist}_\omega(v,t) - \frac{b}{\mathrm{cf}_{\max}} & \text{otherwise,} \end{cases} \tag{5.3}$$

for all $v \in V$ and for arbitrary $b \in [0, M]$. It uses the minimum (unrestricted) driving time as a lower bound on the remaining trip time in case that the current SoC is greater or equal to the minimum energy required to reach the target without recharging. Otherwise, we know that we have to spend some additional time for charging on the way to the target, so we use a bound induced by the weight function ω. Note that we can discard labels whose consumption exceeds M in the search that computes $\mathrm{dist}_c(v,t)$ for all $v \in V$ to save some time in practice. Lemma 5.4 formally proves that the potential function π_ω defined above is consistent.

Lemma 5.4. *The potential function π_ω is consistent.*

Proof. To prove the claim, we show that the reduced costs $\underline{d}(\cdot,\cdot)$ are always nonnegative, i. e., the inequality

$$\underline{d}((u,v),b) = d(u,v) - \pi_\omega(u,b) + \pi_\omega(v, f_{(u,v)}(b)) \geq 0$$

holds for all $(u,v) \in E$ and $b \in [0,M]$. Consider an arbitrary edge $(u,v) \in E$ and assume that the SoC at u is $b \in [0,M]$. We distinguish four cases.

Case 1: $b < \mathrm{dist}_c(u,t)$ and $f_{(u,v)}(b) < \mathrm{dist}_c(v,t)$. The claim follows after a few simple substitutions. We can make use of the fact that $b - c(u,v) \geq f_{(u,v)}(b)$ holds for all $b \in [0,M]$. Consistency then follows directly from the triangle inequality, after performing the simple steps

$$\begin{aligned}
\underline{d}((u,v),b) &= d(u,v) - \pi_\omega(u,b) + \pi_\omega(v, f_{(u,v)}(b)) \\
&= d(u,v) - \mathrm{dist}_\omega(u,t) + \frac{b}{\mathrm{cf}_{\max}} + \mathrm{dist}_\omega(v,t) - \frac{f_{(u,v)}(b)}{\mathrm{cf}_{\max}} \\
&\geq d(u,v) + \frac{c(u,v)}{\mathrm{cf}_{\max}} - \mathrm{dist}_\omega(u,t) + \mathrm{dist}_\omega(v,t) \\
&= \omega(u,v) - \mathrm{dist}_\omega(u,t) + \mathrm{dist}_\omega(v,t) \\
&\geq 0.
\end{aligned}$$

Case 2: $b < \mathrm{dist}_c(u,t)$ and $f_{(u,v)}(b) \geq \mathrm{dist}_c(v,t)$. We make use of both preconditions together with the fact that $b - c(u,v) \geq f_{(u,v)}(b)$ holds for all $b \in [0,M]$ to obtain the inequalities

$$b \geq f_{(u,v)}(b) + c(u,v) \geq \mathrm{dist}_c(v,t) + c(u,v) \geq \mathrm{dist}_c(u,t) > b,$$

which yield a contradiction. Hence, this case cannot occur.

Case 3: $b \geq \mathrm{dist}_c(u,t)$ and $f_{(u,v)}(b) < \mathrm{dist}_c(v,t)$. We know that, due to the triangle inequality, $d(u,v) + \mathrm{dist}_d(v,t) \geq \mathrm{dist}_d(u,t)$ holds. Moreover, $\mathrm{dist}_c(v,t) - f_{(u,v)}(b) > 0$ holds by assumption. We can further exploit that $\mathrm{dist}_\omega(v,t)$ is at least the sum of $\mathrm{dist}_d(v,t)$ and $\mathrm{dist}_c(v,t)/\,\mathrm{cf}_{\max}$ (note that it is possibly greater if the shortest u–t paths in the graph with respect to the cost functions d and c differ). After some substitutions, we thus get

$$\begin{aligned}\underline{d}((u,v),b) &= d(u,v) - \pi_\omega(u,b) + \pi_\omega(v, f_{(u,v)}(b))\\ &= d(u,v) - \operatorname{dist}_d(u,t) + \operatorname{dist}_\omega(v,t) - \frac{f_{(u,v)}(b)}{\mathrm{cf}_{\max}}\\ &\geq d(u,v) - \operatorname{dist}_d(u,t) + \operatorname{dist}_d(v,t) + \frac{\operatorname{dist}_c(v,t)}{\mathrm{cf}_{\max}} - \frac{f_{(u,v)}(b)}{\mathrm{cf}_{\max}}\\ &\geq 0.\end{aligned}$$

Case 4: $b \geq \operatorname{dist}_c(u,t)$ and $f_{(u,v)}(b) \geq \operatorname{dist}_c(v,t)$. This case is trivial; feasibility follows directly from the triangle inequality. □

Observe that the potential function π_ω always evaluates to 0 at the target and overestimates charging speed by construction. Regarding the consistent key, defined for a label ℓ as $\mathrm{key}^*(\ell) = \min_{x\in\mathbb{R}_{\geq 0}} x + \pi(v, f\langle\ell\rangle(x))$, observe that the corresponding terms $x + \operatorname{dist}_d(v,t)$ and $x + \operatorname{dist}_\omega(v,t) - (f\langle\ell\rangle(x)/\,\mathrm{cf}_{\max})$ in Equation 5.3 increase with x (the term $f\langle\ell\rangle(x)/\,\mathrm{cf}_{\max}$ increases with a slope of at most 1). Thus, we obtain the consistent key for the label ℓ by computing the value of $x + \pi_\omega(v, f\langle\ell\rangle(x))$ at the minimum feasible trip time $x_1 := x_{\min}(f\langle\ell\rangle)$ of $f\langle\ell\rangle$ and at the minimum trip time x_2 with $f\langle\ell\rangle(x_2) \geq \operatorname{dist}_c(v,t)$, if it exists. The minimum of both values yields a consistent key, given as $\mathrm{key}^*(\ell) = \{x_1, x_2\}$. Together with Lemma 5.4, this implies correctness of CFP when applying potential shifting with the function π_ω.

Potentials Based on Bound Function Propagation. Even though the potential function π_ω incorporates SoC, lower bounds may be too conservative in that they presume recharging is possible at any time and with the best charging rate. We attempt to be more precise, while keeping computational effort limited, by explicitly constructing lower bound functions.

Again, we run (at query time) a label-correcting search from the given target $t \in V$ on the backward graph $\bar{G}$, but this time computing for each vertex $v \in V$ a *piecewise linear function* $\varphi\colon \mathbb{R} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ mapping SoC to a lower bound on the trip time from v to t. Along the lines of Section 3.1, piecewise linear functions are represented by sequences $\Phi = [(b_1, x_1), \dots, (b_k, x_k)]$ of breakpoints such that $\varphi(b) = \infty$ for $b < b_1$ and $\varphi(b) = x_k$ for $b \geq b_k$. For arbitrary values $b_i \leq b < b_{i+1}$, with $i \in \{1, \dots, k-1\}$, we evaluate the function by linear interpolation as usual. If the sequence Φ of breakpoints is empty, denoted $\Phi = \emptyset$, we obtain $\varphi \equiv \infty$. During the backward search, each vertex stores a *single* label consisting of such a piecewise linear function. To simplify the search, we ignore battery constraints. Hence, domains of bounds are not restricted to $[0, M]$ and we compute (possibly negative) lower bounds on the SoC necessary to reach t. However, we maintain the invariant that all functions are *decreasing* and *convex*

```
   // initialize labels
1  foreach $v \in V$ do
2      $\varphi_v \longleftarrow \emptyset$
3  $\varphi_t \longleftarrow [(0,0)]$
4  $Q$.insert($t$,0)
   // run main loop
5  while $Q$.isNotEmpty() do
6      $u \longleftarrow Q$.deleteMin()
7      if $u \in S$ then
8          $\varphi_u \longleftarrow \text{extend}(\varphi_u, \tilde{\text{cf}}_u)$
9      foreach $(u,v) \in \bar{E}$ do
10         $\varphi \longleftarrow \text{shift}(\varphi_u, [(c(u,v), d(u,v))])$
11         if $\exists x \in \mathbb{R}\colon \varphi(x) < \varphi_v(x)$ then
12             $\varphi_v \longleftarrow \text{merge}(\varphi_v, \varphi)$
13             $Q$.update($v$,key($\varphi_v$))
```

Figure 5.7: Pseudocode of the function propagating potential search for CFP. The algorithm takes as input a (backward) graph $\bar{G} = (V, \bar{E})$ with cost functions $d\colon \bar{E} \to \mathbb{R}_{\geq 0}$ and $c\colon \bar{E} \to \mathbb{R}$, a set $S \subseteq V$ of charging stations, a charging function cf_v for each $v \in S$, and a target vertex $t \in V$. It computes, for each $v \in V$, a piecewise linear function $\varphi_v\colon \mathbb{R} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ that maps SoC to a lower bound on trip time from v to t.

on the interval $[b_1, \infty)$. This greatly simplifies label updates and the computation of functions, which we describe in detail below.

The algorithm resembles (label-correcting) profile search (see Section 3.3.1) and is outlined in Figure 5.7. It is initialized in lines 1–4 with a function φ_t, represented by a single breakpoint $\Phi_t = [(0,0)]$ at the target vertex t, i. e., $\varphi_t(b)$ evaluates to 0 for arbitrary values $b \in \mathbb{R}_{\geq 0}$. All other labels are empty, so their functions always evaluate to ∞. Each step of the algorithm's main loop (lines 5–13) scans a vertex with minimum key (following the generic profile search from Section 3.3.1, the key of a vertex $v \in V$ is the minimum function value $\min_{b \in \mathbb{R}} \varphi_v(b)$ of its label φ_v).

Whenever the search reaches a charging station $u \in S$, we have to ensure that the possibility of recharging is reflected in the label of u and that the function overestimates charging speed. Thus, we *extend* the (tentative) lower bound function φ_u with the charging function $\tilde{\text{cf}}_u$. This results in a new (tentative) lower bound on trip time that incorporates recharging at u and has a slope of at least $-1/\,\text{cf}_{\max}(u)$ (on its subdomain with finite image). Assume we are given a piecewise linear, convex function φ_u at u with breakpoints $\Phi_u = [(b_1, x_1), \dots, (b_k, x_k)]$ that maps SoC to trip time without recharging at u. We obtain the result φ of extending φ_u with $\tilde{\text{cf}}_u$ as follows; see Figure 5.8 for an example. In accordance with our requirements for the potential at u, we

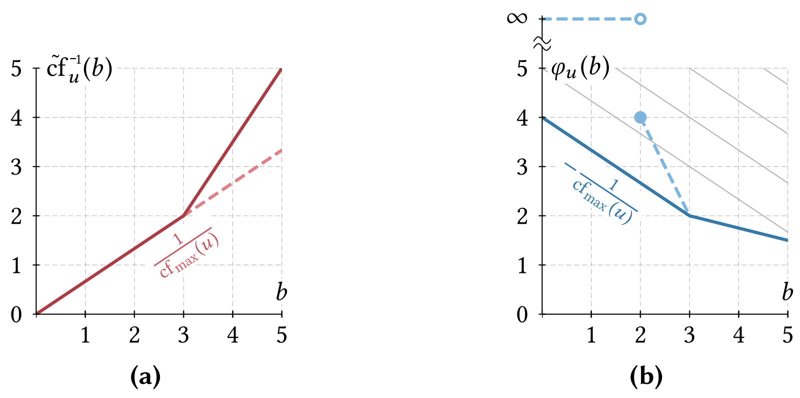


Figure 5.8: Extending functions in the function propagating backward search. (a) The (expanded) inverse charging function $\tilde{\mathrm{cf}}_u^{-1}$ of a charging station $u \in S$ maps SoC to charging time. The dashed line indicates its (inverse) maximum slope, which we use to approximate the (inverse) charging function with a single segment. (b) The lower bound function φ_u (dashed) and the result φ of extending it with the charging function $\tilde{\mathrm{cf}}_u$ (dark blue)

approximate $\tilde{\mathrm{cf}}_u$ with a lower bound given by a single segment with slope $1/\,\mathrm{cf}_{\max}(u)$; see Figure 5.8a. Distributing (lower bounds on) charging time among $\tilde{\mathrm{cf}}_u$ and stations represented by φ_u then corresponds to shifting this segment along the y-axis such that it intersects φ_u; see Figure 5.8b. To find a lower bound on the best possible distribution, let $i \in \{2,\ldots,k-1\}$ be the unique index (if it exists) such that

$$\frac{x_i - x_{i-1}}{b_i - b_{i-1}} \leq -\frac{1}{\mathrm{cf}_{\max}(u)} < \frac{x_{i+1} - x_i}{b_{i+1} - b_i},$$

i. e., the (negative, inverse) maximum slope of $\tilde{\mathrm{cf}}_u$ is at least the slope of the segment from b_{i-1} to b_i, but lower than the slopes of all subsequent segments. We set $i := 1$ if the maximum slope of $\tilde{\mathrm{cf}}_u$ is below the slope of the first segment (from b_1 to b_2), and $i := k$ if the maximum slope of $\tilde{\mathrm{cf}}_u$ is at least the slope of the last segment (from b_{k-1} to b_k). Then, if $b_i \leq 0$, the function φ_u remains unchanged, i. e., $\varphi = \varphi_u$, as charging at u cannot decrease the lower bound in this case. Otherwise, φ is defined by the sequence $\Phi := [(0, x_i + b_i/\,\mathrm{cf}_{\max}(u)), (b_i, x_i), \ldots, (b_k, x_k)]$; see Figure 5.8b. The first segment of this function corresponds to an estimate of the time to charge to an SoC of b_i plus the remaining trip time to t. By construction, φ is convex and overestimates charging speed (provided that the same holds for φ_u).

Since we ignore battery constraints, scanning an outgoing edge $(u,v) \in E$ of u boils down to shifting all breakpoints of the current function φ_u by $c(u,v)$ and $d(u,v)$ on the x- and y-axis, respectively. More formally, given the breakpoints $\Phi_u = [(b_1,x_1),\ldots,(b_k,x_k)]$ of φ_u, we compute a function φ with

$$\Phi := [(b_1 + c(u,v), x_1 + d(u,v)), \ldots, (b_k + c(u,v), x_k + d(u,v))].$$

Then, we check whether the function φ is smaller than the function φ_v in the label of v for at least one $b \in \mathbb{R}$. If this is the case, we *merge* φ and φ_v, i. e., we compute the function defined as their pointwise minimum $\min\{\varphi_v(b), \varphi(b)\}$ for all $b \in \mathbb{R}$. This operation requires a linear-time scan over the breakpoints of both involved functions, similar to the label-correcting profile searches described in Section 3.3.1 and Section 4.2.2. The resulting function φ_v is again piecewise linear, but may no longer be convex. Therefore, we compute, during each merge operation, the *convex lower hull* of the result using Graham's scan [Gra72]. While (slightly) deteriorating the quality of the bound, this reduces the number of breakpoints and simplifies handling of charging stations. We obtain a convex function, which is stored in the label of v. The vertex v is also updated in the priority queue.

It is easy to see that the potential function $\pi_\varphi \colon V \times [0, M] \cup \{-\infty\} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ is consistent when using the computed bounds by setting $\pi_\varphi(v, b) := \varphi_v(b)$ for $v \in V$ and $b \in [0, M]$. Lemma 5.5 proves this formally. The consistent key of a label ℓ at some vertex $v \in V$, with $\text{key}^*(\ell) = \min_{x \in \mathbb{R}_{\geq 0}} x + \pi_\varphi(v, f\langle \ell \rangle(x))$, is computed in a linear scan over $f\langle \ell \rangle$ and the breakpoints of the piecewise linear function φ_v.

Lemma 5.5. *The potential function π_φ is consistent.*

Proof. For an arbitrary edge $(u, v) \in E$, consider the piecewise linear functions φ_u and φ_v at u and v, respectively, after the backward search has terminated. We show that the reduced costs $\underline{d}((u, v), b)$ are nonnegative for all $b \in [0, M]$. We know that φ_u is upper bounded by the result of shifting φ_v by the costs $c(u, v)$ and $d(u, v)$ of the edge (u, v) traversed in backward direction, since this function was merged with φ_u during the search. This implies that $\pi_\varphi(v, b - c(u, v)) + d(u, v) \geq \pi_\varphi(u, b)$ holds for arbitrary $b \in [0, M]$. Moreover, $f_{(u,v)}(b)$ is a lower bound on $b - c(u, v)$, so we have $\pi_\varphi(v, b - c(u, v)) \leq \pi_\varphi(v, f_{(u,v)}(b))$ because π_φ decreases with increasing SoC. Plugging this into the above inequality, we obtain $d(u, v) - \pi_\varphi(u, b) + \pi_\varphi(v, f_{(u,v)}(b)) \geq 0$ for all $b \in [0, M]$, which proves the claim. □

Potential Search on Demand. Computing the potential function for every vertex in the graph is wasteful for short-range queries. To speed up such queries, we run the backward searches that compute vertex potentials *on demand*: Whenever the CFP search requires the potential of some vertex $v \in V$ that was not scanned by the backward search yet, the backward search is executed until v is reached. It is then suspended and only resumed if the potential of another vertex is required that the backward search has not visited. This procedure yields consistent potentials if the backward search is label setting (otherwise, there is no guarantee that a lower bound was computed when a vertex is scanned for the first time). In case of the potential function π_ω, this can be ensured by applying Johnson's shifting technique [Joh77] to its backward searches. For the potential function π_φ, however, the function-propagating search is label correcting, so computing π_φ on demand becomes more involved. We describe modifications to the search and the lower bounds to ensure that π_φ is indeed a consistent potential, even if the search is suspended before it terminates.

First, we can ensure that the minimum key in the priority queue of the backward search is nondecreasing during the course of the algorithm (c. f. Section 4.2.2). After scanning an edge $(u,v) \in E$, consider two functions φ and φ_v corresponding to the result of scanning the edge (u,v) and the current label at v, respectively, before merging these two functions (see line 12 in Figure 5.7). Let φ_v^* denote the result after merging. Since φ_v^* is the convex lower hull of the minimum of φ and φ_v, every breakpoint in φ_v^* must also be contained in φ or φ_v. Let (b,x) be the breakpoint with minimum trip time $x \in \mathbb{R}_{\geq 0}$ contained in the corresponding sequence Φ_v^* of φ_v^*, but not in the sequence Φ_v of φ_v, i. e., $(b,x) \in \Phi_v^*$ and $(b,x) \notin \Phi_v$. If the result of merging improves the label at v, such a point must exist. We set the key of v in the priority queue to the minimum of x and its current key. Since driving time is nonnegative, scanning an edge may only increase the driving time of newly added breakpoints. As a result, propagating breakpoints never decreases the minimum key in the priority queue (see also Section 4.2.2).

Assume that the backward search is suspended at some point and let $x^* \in \mathbb{R}_{\geq 0}$ be current minimum key of the priority queue. For each vertex $v \in V$, consider its current label φ_v. Let φ_v^* denote the function obtained after applying Graham's scan to the result of merging φ_v and the function induced by the single breakpoint $[(0,x^*)]$. We claim that the potential function $\pi_\varphi^*\colon V \times [0,M] \cup \{-\infty\} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ with $\pi_\varphi^*(v,b) := \varphi_v^*(b)$ for all $v \in V$ and $b \in [0,M]$ is consistent. To see this, consider a (multi-)graph $G' = (V,E')$ constructed from the input graph G by adding, for every $v \in V$, an edge (v,t) with driving time $d(v,t) := x^*$ and consumption $c(v,t) := 0$. It is easy to verify that the potential function π_φ^* on G is equivalent to the potential function π_φ on G'. Hence, π_φ^* is a consistent potential function for G'. Observe that this implies that π_φ^* is also consistent on G, since reduced edge costs must be nonnegative for the subset $E \subseteq E'$. Lemma 5.6 follows immediately from this observation and Lemma 5.5.

Lemma 5.6. *The potential function π_φ^* is consistent.*

Note that we have to keep the original function φ_v after suspending the backward search, in case it is resumed later. Hence, we do not store π_φ^* explicitly, but perform the necessary merge operation and Graham's scan on demand when the potential is requested. Given that keys of labels are consistent and by Lemma 5.4, Lemma 5.5, and Lemma 5.6, we obtain Theorem 5.7, which summarizes our findings on A* Search.

Theorem 5.7. *The CFP algorithm computes the correct output when using either of the potential functions π_ω, π_φ, or π_φ^*.*

Implementation Details. When using potentials on demand, we can suspend the backward search at any time and derive lower bounds for CFP. However, bound quality of π^*_φ may deteriorate if the search is suspended too early, as it depends on the current minimum key in the priority queue of the backward search. Therefore, we do not abort the search immediately when a vertex $v \in V$ in question is scanned for the first time, but continue until the minimum key x^* in the priority queue is significantly greater than the key induced by the first breakpoint (b_1, x_1) of φ_v (in our experiments, we suspend the search if $x^* > \min\{2x_1, x_1 + 3\,600\}$, where time is measured in seconds).

5.2.5 Contraction Hierarchies

Using an offline preprocessing step, CH [Gei+12b] iteratively contract the vertices of the input graph and add *shortcuts* in the remaining graph to retain correct distances. These shortcuts then help reducing the search space in online queries (see Section 3.3.2 for details). When adapting CH to our scenario, vertex contraction becomes more expensive, as each shortcut represents a pair consisting of driving time and an SoC profile. Moreover, we need a shortcut for every nondominated path. Hence, the resulting search graph may contain multi-edges.

We compute a *partial* CH, i. e., we contract only some vertices (the *component*), leaving an uncontracted *core* graph—a common approach in complex scenarios [DPW15b, Kle+17, Sto12a]. As in Section 4.3.4, we keep all charging stations in the graph uncontracted. Thus, complexity induced by charging stations only is contained within the core (simplifying the search in the component). Shortcuts store the driving time and the SoC profile (represented by three values as described in Section 4.1.3) of the path that they represent.

Witness Search. During preprocessing, we perform *witness searches* when contracting a vertex, to test whether all shortcut candidates are necessary to maintain distances in the current overlay. Given a shortcut candidate (u, v) with $u \in V$ and $v \in V$, we run a variant of the BSP algorithm that propagates labels consisting of the driving time and the SoC profile of a path (represented by three values), starting from u. A label *dominates* another label in this search if its driving time is smaller or equal to the driving time of the other label and its SoC profile dominates that of the other label. Keys in the priority queue follow a lexicographic order of the labels. The witness search stops if either the shortcut candidate is dominated by a label at v or the minimum key in the priority queue exceeds the key induced by the shortcut candidate.

In order to reduce preprocessing time, we simplify the witness search as follows. First, we only search for single witnesses that dominate a shortcut candidate. In other words, we only perform pairwise comparisons between labels at the head of a shortcut candidate and the candidate itself. Thereby, we might insert an unnecessary shortcut whose SoC profile is dominated by the upper envelope of multiple SoC profiles corresponding to labels with lower driving time. Second, during the witness search, we limit the number of labels per vertex to a small constant (10 in our experiments).

Whenever this size is exceeded, we identify (in a linear scan over the sorted labels) the pair of labels that has the minimum difference in terms of driving time. Of these two labels, we remove the one with smaller difference to its next closest label (in order to keep the gap between the remaining labels small). Finally, we prune the search after a fixed hop limit [Gei+12b] (20 in our experiments). Taking these measures, we may possibly insert unnecessary shortcuts. Thus, queries may slow down slightly, but correctness is not affected.

Queries. Since we compute a partial CH, the query algorithm consists of two phases. Given a source $s \in V$, a target $t \in V$, and the initial SoC $b \in [0, M]$, the first phase runs a backward CH search from t, scanning only upward edges with respect to the vertex order. This search operates on the component, so it is pruned at core vertices (i. e., outgoing edges of core vertices are not scanned). As the component contains no charging stations, a basic variant of the BSP algorithm suffices. Note, however, that the SoC at t is yet unknown and therefore, the search algorithm computes SoC profiles instead of SoC values (as in witness search). For every nondominated label at any vertex visited by the search, we add a (temporary) shortcut from this vertex to the target. The second phase runs CFP from s and is restricted to upward edges, core edges, and the temporary edges added by the backward search.

Implementation Details. During preprocessing of CH, the next vertex in the contraction order is determined from the measures Edge Difference (ED), Deleted Neighbors (DN), and Cost of Queries (CQ) [Gei+12b]. The priority of a vertex (higher priority corresponds to a higher rank; see Section 3.3.2) is then set to $64\,\mathrm{ED} + \mathrm{DN} + \mathrm{CQ}$. To reduce the number of witness searches, we cache shortcuts computed during the computation of ED. This requires a simulated contraction; see Geisberger et al. [Gei+12b]. Whenever witness searches for multiple shortcuts with the same source are required during contraction of some vertex, we run a single multi-target search instead. Further, to improve query times, we reorder vertices after preprocessing, such that core vertices are in consecutive memory.

5.2.6 CHArge

Combining CH and A* search (restricting A* search to the core), we obtain our fastest exact algorithm, *CHArge (CH, A*, Charging Stops)*. The query algorithm consists of three phases, namely, a unidirectional (backward) phase from the target $t \in V$ in the component to add temporary shortcuts, a backward search in the (much smaller) core enriched with temporary shortcuts to compute a potential function (either π_ω or π^*_φ), and a forward phase running CFP (augmented with A* search) from the source $s \in V$, which is restricted to upward edges, core edges, and temporary edges. Potentials of component vertices are set to 0 for this search. Observe that consistency of the

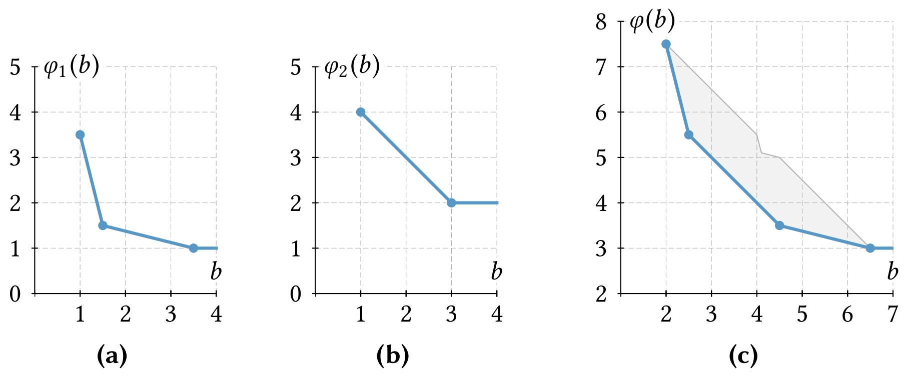


Figure 5.9: Linking piecewise linear lower bound functions. Linear segments between indicated breakpoints show the (finite) values of two functions and the result of linking them. (a) The function φ_1 is defined by three breakpoints. (b) The function φ_2 is defined by two breakpoints. (c) Linking φ_1 and φ_2 yields the function φ. It is the lower envelope of the shaded area, which corresponds to (finite) values of $\varphi_1(b^*) + \varphi_2(b - b^*)$ for different choices of $b^* \in \mathbb{R}$.

resulting potential function is not violated, since there are no edges pointing from the core into the component. As described in Section 5.2.4, potentials in the core can be computed on demand, in which case the second and third phase are interweaved and their searches are executed alternately.

Computing Potentials in the Core. To decrease running time of the second phase, we precompute lower bounds of core shortcuts for the potential function π^*_φ, i. e., for each (ordered) pair $u \in V$ and $v \in V$ of vertices connected by at least one shortcut, we compute a decreasing and convex piecewise linear function that yields a lower bound on driving time from u to v for a given SoC. To this end, we perform Graham's scan [Gra72] on all pairs $[(b,x)]$ of minimum required SoC $b \in [0,M]$ and driving time $x \in \mathbb{R}_{\geq 0}$ corresponding to some shortcut edge between u and v (recall that the component may contain multi-edges). However, this also requires us to adapt the function-propagating search in the second phase, since scanning an edge no longer consists of simply shifting a function by two constant values (c. f. line 10 of Figure 5.7 in Section 5.2.4). Instead, piecewise linear functions of labels have to be *linked* with shortcut edges, which are represented by piecewise linear functions as well. In what follows, we describe how this can be done in linear time in the number of breakpoints of both functions.

Consider piecewise linear functions $\varphi_1 \colon \mathbb{R} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ and $\varphi_2 \colon \mathbb{R} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ mapping SoC to lower bounds on the trip time along two paths in the graph, such that both are decreasing and convex on their subdomain with finite image. Let the two functions be given by their respective sequences $\Phi_1 = [(b_1^1, x_1^1), \dots, (b_1^k, x_1^k)]$ and $\Phi_2 = [(b_2^1, x_2^1), \dots, (b_2^\ell, x_2^\ell)]$ of breakpoints. The *link operation* takes the functions φ_1

and φ_2 as input and computes a function φ that reflects the concatenation of both paths. Hence, given an arbitrary SoC $b \in [0, M]$, the value $\varphi(b)$ is a lower bound on the trip time when traversing the paths represented by φ_1 and φ_2, such that overall consumption does not exceed the SoC b. To this end, the link operation identifies values $b_1 \in \mathbb{R}$ and $b_2 \in \mathbb{R}$, such that $b_1 + b_2 = b$ and $\varphi_1(b_1) + \varphi_2(b_2)$ is minimized. Hence, we seek to compute the function $\varphi \colon \mathbb{R} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ with

$$\varphi(b) := \min_{b^* \in \mathbb{R}} \varphi_1(b^*) + \varphi_2(b - b^*), \tag{5.4}$$

which yields the desired lower bound on the trip time for traversing φ_1 and φ_2. Figure 5.9 shows an example. Below, we describe an algorithm that computes a sequence Φ of breakpoints to represent this function φ. Afterwards, we prove its correctness.

Starting with an empty sequence $\Phi = \emptyset$, the link operation iteratively appends breakpoints to Φ, each of which is the sum of two breakpoints from Φ_1 and Φ_2. For $b = b_1^1 + b_2^1$ there exists exactly one value $b^* = b_1^1$ in Equation 5.4 that yields a finite trip time. We obtain $\varphi(b_1^1 + b_2^1) = x_1^1 + x_2^1$ and the first breakpoint of Φ is $(b_1^1 + b_2^1, x_1^1 + x_2^1)$. For subsequent breakpoints, the basic idea is to follow the function that offers the better (i. e., lower) slope. Assume that the previous breakpoint added to Φ is $(b_1^i + b_2^j, x_1^i + x_2^j)$ for some $i \in \{1, \dots, k\}$ and $j \in \{1, \dots, \ell\}$. Consider the slope

$$\sigma_1^i := \begin{cases} \frac{b_1^{i+1} - b_1^i}{x_1^{i+1} - x_1^i} & \text{if } i < k, \\ 0 & \text{otherwise,} \end{cases}$$

of the next segment of the function φ_1. Let the slope σ_2^j be defined symmetrically. Then the next breakpoint is $(b_1^{i+1} + b_2^j, x_1^{i+1} + x_2^j)$ if $\sigma_1^i < \sigma_2^j$ and $(b_1^i + b_2^{j+1}, x_1^i + x_2^{j+1})$ if $\sigma_1^i > \sigma_2^j$. In other words, we pick the next breakpoint of the currently steeper function (keeping the same point as before for the other function). In the special case $\sigma_1^i = \sigma_2^j$ we obtain three collinear points, so the next breakpoint is $(b_1^{i+1} + b_2^{j+1}, x_1^{i+1} + x_2^{j+1})$. The scan is stopped as soon as the last point $(b_1^k + b_2^\ell, x_1^k + x_2^\ell)$ is reached and added to Φ.

Clearly, the scan described above runs in linear time in the number $k + \ell$ of breakpoints of φ_1 and φ_2. Moreover, as segments are appended in increasing order of original slope, the resulting function φ is also decreasing and convex. Lemma 5.8 formally proves that the link operation in fact computes the correct result.

Lemma 5.8. *Let $\varphi_1 \colon \mathbb{R} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ and $\varphi_2 \colon \mathbb{R} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ be piecewise linear functions defined by sequences $\Phi_1 = [(b_1^1, x_1^1), \dots, (b_1^k, x_1^k)]$ and $\Phi_2 = [(b_2^1, x_2^1), \dots, (b_2^\ell, x_2^\ell)]$ of breakpoints, respectively, such that both functions are decreasing and convex on their subdomains $[b_1^1, \infty)$ and $[b_2^1, \infty)$ with finite image. The link operation described above computes a sequence Φ of breakpoints that corresponds to a function $\varphi \colon \mathbb{R} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ with $\varphi(b) = \min_{b^* \in \mathbb{R}} \varphi_1(b^*) + \varphi_2(b - b^*)$ for all $b \in \mathbb{R}$.*

Proof. For $b < b_1^1 + b_2^1$, there exists no value $b^* \in \mathbb{R}$ such that $\varphi_1(b^*)$ and $\varphi_2(b - b^*)$ are both finite, so the result of linking equals ∞. For $b = b_1^1 + b_2^1$ there is exactly one such value $b^* \in \mathbb{R}$ that yields a finite trip time and we obtain $\varphi(b_1^1 + b_2^1) = x_1^1 + x_2^1$, which corresponds to the first breakpoint of Φ.

Let $(b_1^i + b_2^j, x_1^i + x_2^j)$ with $i \in \{1, \dots, k\}$ and $j \in \{1, \dots, \ell\}$ denote the last point that was added to Φ. Without loss of generality, assume that the next segment of the function φ_1 is at least as steep as the next segment of φ_2, i. e., $\sigma_1^i \leq \sigma_2^j$. Hence, our algorithm sets $(b_1^{i+1} + b_2^j, x_1^{i+1} + x_2^j)$ as the next breakpoint of Φ (or adds a segment that contains this point in the special case $\sigma_1^i = \sigma_2^j$). Thus, the slope of φ is σ_1^i for all $b \in [b_1^i + b_2^j, b_1^{i+1} + b_2^j]$. To prove the claim, we show that

$$\varphi(b) = x_1^i + x_2^j + \sigma_1^i(b - b_1^i - b_2^j) \leq \varphi_1(b^*) + \varphi_2(b - b^*)$$

holds for all $b \in [b_1^i + b_2^j, b_1^{i+1} + b_2^j]$ and $b^* \in (-\infty, b]$. Since both φ_1 and φ_2 are convex, we know that $\varphi_1(b) \geq x_1^i + \sigma_1^i(b - b_1^i)$ and $\varphi_2(b) \geq x_2^j + \sigma_2^j(b - b_2^j)$ hold for all $b \in \mathbb{R}$. This immediately yields

$$\begin{aligned}
\varphi_1(b^*) + \varphi_2(b - b^*) &\geq x_1^i + \sigma_1^i(b^* - b_1^i) + x_2^j + \sigma_2^j(b - b^* - b_2^j) \\
&\geq x_1^i + \sigma_1^i(b^* - b_1^i) + x_2^j + \sigma_1^i(b - b^* - b_2^j) \\
&= x_1^i + x_2^j + \sigma_1^i(b - b_1^i - b_2^j) \\
&= \varphi(b).
\end{aligned}$$

Since $\varphi(b) \geq \min_{b^* \in \mathbb{R}} \varphi_1(b^*) + \varphi_2(b - b^*)$ must hold for all $b \in \mathbb{R}$ by construction, this completes our proof. □

5.2.7 Heuristic Approaches

With an $\mathcal{NP}$-hard problem at hand, we propose heuristic approaches based on CHArge, which drop optimality to reduce query times. Their basic idea is as follows. During the third phase of CHArge (running the CFP algorithm on the core graph), whenever the search scans multiple shortcuts (u, v) between two vertices $u \in V$ and $v \in V$, at most one new label is added to $L_{\text{uns}}(v)$. This saves time for dominance checks and label insertion in the label set $L_{\text{uns}}(v)$. We use the potential at v to determine a shortcut that minimizes the key of the new label (i. e., the trip time from the source $s \in V$ to v plus a lower bound on the trip time from v to the target $t \in V$), and add only this label to $L_{\text{uns}}(v)$. Recall that the potential depends on the SoC at v, hence scanning different shortcuts may result in different potentials at v.

Our first heuristic, denoted CHArge-H$_\varphi$, uses the potential function π_φ to determine the best shortcut. Given a label ℓ at some vertex $u \in V$, scanning an outgoing shortcut (u, v) to a vertex $v \in V$ results in some label ℓ' at v. We compute its key, which minimizes the sum $x + \pi_\varphi(v, f\langle\ell'\rangle(x))$ for arbitrary $x \in \mathbb{R}_{\geq 0}$ (as described in

Section 5.2.4). However, the label is only added to $L_{\text{uns}}(v)$ if this key is minimal among all labels at v constructed within the current vertex scan.

The idea of our second heuristic, denoted CHArge-H$_\omega$, is to use the potential function π_ω instead of π_φ. Additionally, when identifying the only label to be inserted into the set $L_{\text{uns}}(v)$ of a vertex $v \in V$, we ignore battery constraints and presume that we are not close to the target, i. e., that we are in the case $b < \text{dist}_c(v,t)$ of Equation 5.3 for arbitrary SoC $b \in [0,M]$. Then the best shortcut (u,v) does not depend on the SoC at v, but only on the distance $\text{dist}_\omega(v,t)$. Hence, we can precompute the optimal shortcut for each pair of neighbors $u \in V$ and $v \in V$, namely, the one that minimizes $\omega(u,v)$. During a query, instead of scanning all shortcuts, we always use the precomputed shortcut for each neighbor v of u.

A third, even more aggressive variant, which we denote by CHArge-H$^{\text{A}}_\omega$, uses the same idea as in CHArge-H$_\omega$ already during vertex contraction for CH, keeping only the *optimal* shortcut with respect to the cost function ω for each pair of vertices. Thus, we no longer allow the creation of multi-edges during preprocessing. This significantly reduces the total number of shortcuts in the core graph, allowing the contraction of further vertices. While the resulting search graph can no longer be used for exact queries, CHArge-H$^{\text{A}}_\omega$ is capable of answering heuristic queries much faster. The query algorithm of CHArge-H$^{\text{A}}_\omega$ is identical to CHArge-H$_\omega$, however, solutions may differ as it operates on a sparser graph.

Despite their heuristic nature, it is actually possible to formally grasp under which circumstances the heuristics CHArge-H$_\omega$ and CHArge-H$^{\text{A}}_\omega$ use an optimal shortcut [Zun14]. Basically, we know that if charging is inevitable and charging at a rate of $\text{cf}_{\max}$ is possible when needed, then $\text{dist}_\omega(\cdot,\cdot)$ yields a *tight* bound on the remaining trip time. The following Proposition 5.9 formalizes this insight. Recall that when computing shortcut edges (u,v) for two vertices $u \in V$ and $v \in V$, the only possible charging stations on the underlying u–v path are u and v (see Section 5.2.5).

Proposition 5.9. *Given two vertices $u \in V$ and $v \in V$, a u–v path P that contains no charging stations (except possibly u and v) and minimizes the cost $\omega(P)$ among all u–v paths is a subpath of a fastest feasible s–t path from a given source $s \in V$ to a given target $t \in V$ if the following conditions are met.*

1. *The fastest feasible s–t path contains u and v in this order, but no charging station on the subpath from u to v (except u and v).*
2. *The SoC at u is not sufficient to reach t without recharging.*
3. *The SoC never reaches the capacity M (such that battery constraints need to be applied) on the path from u to the next charging station that is used.*
4. *There is a charging station available on the subpath from v to t before the battery runs out and the uniform charging speed at this station is $\text{cf}_{\max}$.*

5.3 Integrating Adaptive Speeds

So far, all algorithms discussed in this chapter find the fastest *route* subject to battery constraints. Yet in reality, travel time and energy consumption are not only affected by the choice of the route itself, but also by *driving behavior.* Assuming a single, fixed speed per road segment neglects solutions that may save energy by reducing driving speed when necessary. Allowing *multiple* driving speeds (and consumption values) per road segment, one could, e. g., save energy on the motorway by driving at reasonable speeds below the posted speed limits. In this section, we consider continuous, *adaptive speeds*, i. e., we allow the EV to adjust its speed within reasonable limits to reach the destination as fast as possible and with sufficient SoC. Hence, in addition to the actual route from the source to the target, we also have to specify (optimal) driving speeds along that route. In practice, these can be passed to the driver as recommendations or directly to a cruise control unit. With the advent of autonomous vehicles, the output of our algorithms can also be used for speed planning of self-driving EVs, either directly or after further refinement [Flo+15].

In our extended problem setting, the same road segment can be passed at different speeds (we omit stops at charging stations in this section, though). A straightforward way to model these options is to *sample* reasonable speeds for each road segment [Bau+14, GP14, HF14, SMS17]. Then, one can add *parallel* edges in the underlying graph representation, which correspond to alternative driving speeds (inducing certain values of driving time and energy consumption); see Figure 5.10. The major benefit of this approach is its simplicity: We can immediately apply the basic BSP algorithm described in Section 5.1 to solve the extended problem. However, it also comes with several drawbacks. First of all, parallel edges greatly increase running time, due to a larger number of nondominated solutions. In fact, the number of nondominated (i. e., Pareto-optimal) solutions can be exponential even on a single route; see Figure 5.10a. Consequently, only heuristic algorithms achieve practical running times [Bau+14, GP14, HF14]. By discretizing a continuous range of tradeoffs, parallel edges that model alternative speeds have other undesirable effects, such as producing many insignificant, yet nondominated solutions. Figure 5.10a shows such an example, where some nondominated solutions at the target vertex t provide rather unattractive tradeoffs, namely, spending ten extra units of time to save only one unit of energy. Adding another sample (indicated by the dashed edge), on the other hand, results in a new label at t that dominates one of these less favorable solutions. In other words, the number of samples influences both running time and result quality: More samples increase running time, but fewer samples reduce quality. Similarly, adding or contracting degree-two vertices in the graph, which are commonly included for visualization purposes, affects the solution space even if distances are maintained; see Figure 5.10b. This is clearly not desirable, since such modeling decisions should not have any impact on the optimal solution.

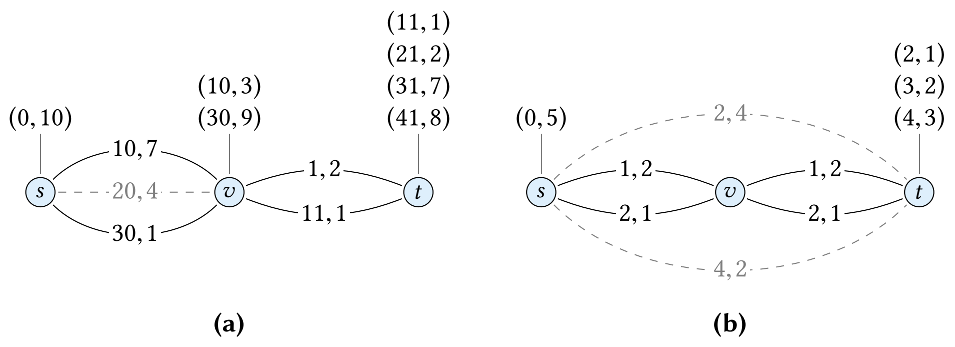


Figure 5.10: Adaptive speeds modeled as parallel edges. Edge labels indicate tuples of driving time and energy consumption. We also show the label sets of vertices as computed by the BSP algorithm in an s–t query (ignoring dashed edges). (a) The initial SoC is 10. Adding the dashed edge (s, v) results in a new label $(20, 6)$ at v and a label $(21, 4)$ at t, which dominates the label $(21, 2)$ in the current set. (b) The initial SoC is 5. Contraction of v results in the dashed edges. Although distances are maintained, the algorithm no longer computes the label $(3, 2)$ on the modified input.

To remedy the above issues, we propose a more sophisticated model, which uses continuous *functions* to model the tradeoffs on edges [HF14]. Using realistic consumption models, we obtain a nonlinear function for each road segment, mapping driving time to energy consumption (Section 5.3.1). Then, we derive operations to compute tradeoff functions representing paths instead of single road segments (Section 5.3.2). Using these basic operations, we describe a generalization of the (exponential-time) BSP algorithm to our problem setting and discuss improvements for better running times (Section 5.3.3). To further reduce query times, we incorporate techniques based on A* search (Section 5.3.4) and CH (Section 5.3.5). Both approaches can be combined to achieve best performance in practice.

5.3.1 Model and Problem Statement

Instead of single scalar values $d(e)$ and $c(e)$ for driving time and energy consumption of an edge $e \in E$ in the input graph $G = (V, E)$, we assume that there is a *tradeoff function* $g_e \colon \mathbb{R}_{>0} \to \mathbb{R}$ based on a physical consumption model, mapping the desired driving time $x \in \mathbb{R}_{>0}$ along the edge e to the resulting energy consumption $g_e(x)$. In reality, the driving time on a road segment cannot be chosen arbitrarily. Lower bounds are induced by speed limits and the maximum speed of the vehicle. On the other hand, driving slower than a reasonable minimum speed would mean to become an obstacle for other drivers. Additionally, there is a certain point at which driving slower will no longer pay off in terms of energy consumption. This yields (positive) minimum and maximum driving times $\underline{\tau}_e \in \mathbb{R}_{>0}$ and $\bar{\tau}_e \in \mathbb{R}_{>0}$, respectively, for the function g_e, with $\underline{\tau}_e \leq \bar{\tau}_e$. We incorporate these bounds into a *consumption function* $c_e \colon \mathbb{R}_{\geq 0} \to \mathbb{R} \cup \{\infty\}$, which is given as

$$c_e(x) := \begin{cases} \infty & \text{if } x < \underline{\tau}_e, \\ g_e(\bar{\tau}_e) & \text{if } x > \bar{\tau}_e, \\ g_e(x) & \text{otherwise.} \end{cases} \tag{5.5}$$

Thus, driving times below $\underline{\tau}_e$ are infeasible (modeled as infinite consumption) and driving times above $\bar{\tau}_e$ become unprofitable. A driving time $x \in \mathbb{R}_{\geq 0}$ is also called *admissible* if $x \in [\underline{\tau}_e, \bar{\tau}_e]$, i. e., it lies in the relevant subdomain of c_e. In the special (degenerate) case $\underline{\tau}_e = \bar{\tau}_e$, the function c_e represents a constant pair $(\underline{\tau}_e, c_e(\underline{\tau}_e))$ of fixed driving time and energy consumption. We call c_e *constant* in this case, as the edge e allows no speed adaptation.

As before, we assume that the EV is equipped with a battery that has a certain *capacity* $M \in \mathbb{R}_{\geq 0}$ and that its SoC must not drop below 0 nor exceed M. When incorporating these constraints into our setting, we obtain a *bivariate* SoC function $f_e \colon \mathbb{R}_{\geq 0} \times [0, M] \cup \{-\infty\} \to [0, M] \cup \{-\infty\}$ for every edge $e = (u, v) \in E$, mapping the SoC at u to the resulting SoC at v when traversing e with a specific driving time. The function f_e is given by

$$f_e(x, b) := \begin{cases} -\infty & \text{if } b - c_e(x) < 0, \\ M & \text{if } b - c_e(x) > M, \\ b - c_e(x) & \text{otherwise,} \end{cases} \tag{5.6}$$

where an SoC of $-\infty$ denotes an empty battery. Hence, $f_e(x, b) = -\infty$ implies that the edge cannot be traversed at the corresponding speed (as it would cause the battery to run empty). Note that we obtain the SoC function f_e by pointwise application of battery constraints (see Section 4.1.1) to the consumption function c_e of e.

Given the SoC $b_s \in [0, M]$ at a source $s \in V$, we can determine a corresponding SoC $b_t \in [0, M] \cup \{-\infty\}$ at a target $t \in V$ if we compute an s–t path $[s = v_1, \dots, v_k = t]$ and pick driving times $x_i \in \mathbb{R}_{\geq 0}$ for all edges (v_i, v_{i+1}) of the path, with $i \in \{1, \dots, k-1\}$. Starting at the source $s = v_1$, the SoC at $v_k = t$ is obtained after iteratively evaluating the SoC function $f_{(v_i, v_{i+1})}$ at x_i and the SoC at the previous vertex v_i. Formally, we then get

$$b_t = f_{(v_{k-1}, t)}(x_{k-1}, f_{(v_{k-2}, v_{k-1})}(\dots f_{(s, v_2)}(x_1, b_s) \dots)).$$

Note that $b_t = -\infty$ holds if the path is infeasible (for the given driving times). Due to physical constraints, we presume that for cycles this procedure never raises the SoC at $s = t$. In other words, minimum values $c_e(\bar{\tau}_e)$ of consumption functions c_e of edges $e \in E$ must not induce cycles with negative energy consumption in the graph.

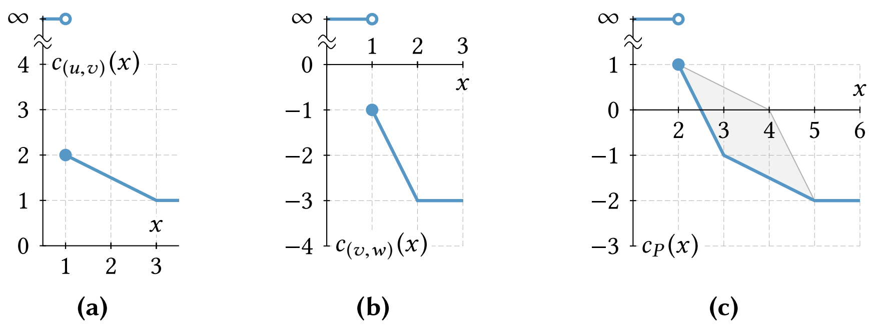


Figure 5.11: Consumption functions in a simplistic model. (a) Consumption function $c_{(u,v)}$ of an edge (u,v) with minimum driving time $\underline{\tau}_{(u,v)} = 1$ and maximum driving time $\bar{\tau}_{(u,v)} = 3$. (b) Consumption function $c_{(v,w)}$ of an edge (v,w) with minimum driving time $\underline{\tau}_{(v,w)} = 1$ and maximum driving time $\bar{\tau}_{(v,w)} = 2$. (c) The consumption function c_P of the path $P = [u,v,w]$. The shaded area indicates possible pairs of driving time and consumption along the path.

Given a source vertex $s \in V$, a target vertex $t \in V$, and an initial SoC $b_s \in [0,M]$, we seek an s–t path $P = [s = v_1, v_2 \dots, v_k = t]$ together with driving times x_i for every edge (v_i, v_{i+1}) of P, where $i \in \{1,\dots,k-1\}$, such that battery constraints are respected and the overall driving time $x := \sum_{i=1}^{k-1} x_i$ is minimized. An instance of our problem where all functions are degenerate constant tuples is also an input instance to the $\mathcal{NP}$-hard problem introduced in Section 5.1. Hence, the extended problem we consider in this section is $\mathcal{NP}$-hard as well.

To gain insights about the structure of optimal solutions, we now derive consumption functions and (bivariate) SoC functions for given *paths* instead of edges. We illustrate such functions in an example using simplistic but vivid tradeoff functions. Afterwards, we propose a more realistic model, which is used in the remainder of this section.

A Simplified Model. We illustrate consumption functions and SoC functions and examine their complexity for a rather simplistic consumption model. For now, assume that the tradeoff function g_e of every edge $e \in E$ is *decreasing* and *linear*, i. e., we have $g_e(x) = \alpha x + \beta$ for all $x \in \mathbb{R}_{\geq 0}$, where $\alpha \in \mathbb{R}_{\leq 0}$ and $\beta \in \mathbb{R}$ are constant coefficients. The values α and β may differ between edges, though, to reflect different road types or other relevant factors [BE05, Yao+13]. Figure 5.11a and Figure 5.11b show corresponding consumption functions (plugging in limits $\underline{\tau}_{(v,w)}$ and $\bar{\tau}_{(v,w)}$ on driving time) for two edges (u,v) and (v,w). We are interested in the consumption function of the path $P = [u,v,w]$, i. e., a function c_P that maps driving time $x \in \mathbb{R}_{\geq 0}$ spent on the u–w path to the *minimum* energy consumption $c_P(x)$ on the path. Formally, to get the value of $c_P(x)$ for some driving time $x \in \mathbb{R}_{\geq 0}$, we have to pick (nonnegative) values $x_1 \in \mathbb{R}_{\geq 0}$ and $x_2 \in \mathbb{R}_{\geq 0}$, such that $x = x_1 + x_2$ and $c_{(u,v)}(x_1) + c_{(v,w)}(x_2)$ is minimized. The shaded area in Figure 5.11c indicates possible distributions of driving

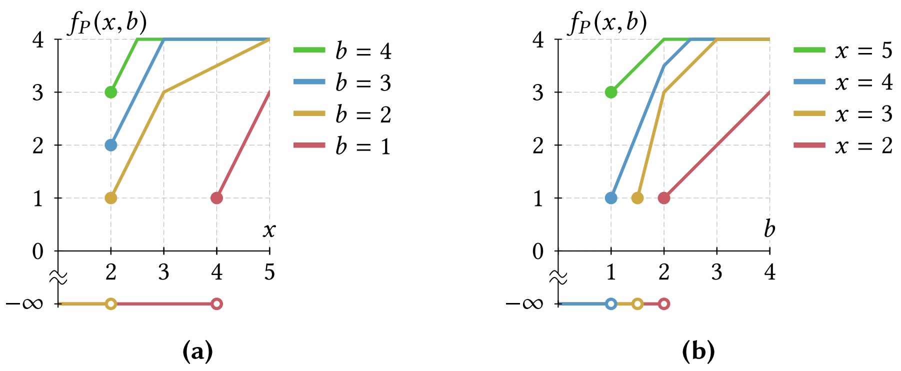


Figure 5.12: The bivariate SoC function of the path P from Figure 5.11, assuming a battery capacity of $M = 4$. (a) The SoC $f_P(x,b)$ at w, subject to driving time $x \in \mathbb{R}_{\geq 0}$ on P for different fixed values $b \in \{1,2,3,4\}$ of initial SoC. (b) The SoC $f_P(x,b)$ at w, subject to initial SoC $b \in [0,M]$ for different fixed driving times $x \in \{2,3,4,5\}$.

times x_1 and x_2 among the two edges and the resulting energy consumption. The lower envelope of this area yields the desired function c_P; see Lemma 5.8 in Section 5.2.6 for a formal proof. Intuitively, we want to spend as much of the available extra time (exceeding the minimum 2) as possible on the edge that provides the best tradeoff, i. e., the consumption function with the steeper slope (where spending additional time saves most energy). As a result, the consumption function of a path is always *convex* on the subdomain where its image is finite; see Figure 5.11c. Observe that, while tradeoff functions of single edges $e \in E$ are linear on the interval $[\underline{\tau}_e, \bar{\tau}_e]$ of admissible driving times, the tradeoff function of a path is *piecewise* linear on the interval induced by its minimum and maximum driving time. The number of linear subfunctions defining the function c_P is bounded by the number of edges in the path [And15].

The situation becomes more involved if we also take battery constraints into account. Then, energy consumption not only depends on the driving time we are willing to spend along a path, but also on the initial SoC. Hence, we obtain a bivariate function f_P, which maps driving time and initial SoC at u to the SoC at w. Note that consumption is positive on the first edge (u,v) and negative on the second edge (v,w) in Figure 5.11. As before, the edge (v,w) provides the better tradeoff. However, for low initial SoC, we have to ensure that the first edge (u,v) can be traversed. Hence, spending some additional time on this edge may be inevitable to obtain a feasible solution. In contrast, high SoC values may prevent recuperation along the second edge (v,w), so driving slower no longer pays off at some point. Figure 5.12 sketches the resulting bivariate SoC function for specific values of initial SoC and driving time. For a given initial SoC $b \in [0,M]$ at u, we see how spending more time on the path can increase the SoC at w (Figure 5.12a). When fixing the driving time $x \in \mathbb{R}_{\geq 0}$ (Figure 5.12b), the optimal amount of time spent on each edge varies with the initial SoC at u. Consequently, the

SoC function of a path P no longer has the specific form as in the case of scalar edge costs, even if we fix the total driving time spent on the path (c. f. Lemma 4.1).

Below, we propose more realistic (nonlinear) tradeoff functions, which we use in the remainder of this section. Although these realistic functions require a more technical analysis, many observations made for our simplistic (linear) model carry over to the more realistic (nonlinear) tradeoff functions.

A Realistic Model. Both considered metrics, driving time and energy consumption, depend on the vehicle's speed. In accordance with realistic physical models established in the literature [Agr+16, Asa+16, Bed+16, FAR16, HF14, LL12, Lv+16], we assume that energy consumption on a certain road segment $e \in E$ can be expressed by a function $h_e \colon \mathbb{R}_{>0} \to \mathbb{R}$ given as

$$h_e(v) = \lambda_1 v^2 + \lambda_2 s_e + \lambda_3, \tag{5.7}$$

where $v \in \mathbb{R}_{>0}$ is the (constant) vehicle speed, $s_e \in \mathbb{R}$ is the (constant) slope of the road segment, and $\lambda_1 \in \mathbb{R}_{\geq 0}$, $\lambda_2 \in \mathbb{R}_{\geq 0}$, and $\lambda_3 \in \mathbb{R}_{\geq 0}$ are constant nonnegative coefficients of the consumption model. (The term $\lambda_1 v^2$ is caused by aerodynamic drag, which increases with driving speed in a superlinear fashion.) Note that energy consumption can become negative for downhill segments with $s_e < 0$. The parameters λ_1, λ_2, and λ_3 may vary for different edges due to, e. g., different road types or other factors affecting energy consumption [BE05, SHS11, Yao+13]. Assuming constant speed and slope per edge is not a restriction, since we can add intermediate vertices in the graph to model changing conditions. Furthermore, one can show that deliberately varying the speed along a single road segment (with constant slope and speed limit) never pays off in our model [HF14, Corollary 1].

Since we are interested in functions mapping *driving time* $x \in \mathbb{R}_{>0}$ to energy consumption $g_e(x)$, we substitute $v = \ell_e/x$ in Equation 5.7, where $\ell_e \in \mathbb{R}_{\geq 0}$ denotes the length of the road segment. As slope and length of an edge are fixed, we simplify this below by setting $\alpha := \lambda_1 \ell_e^2$ and $\gamma := \lambda_2 s_e + \lambda_3$. Observe that $\alpha \in \mathbb{R}_{\geq 0}$ is nonnegative, while $\gamma \in \mathbb{R}$ may have negative values (for downhill edges). We introduce a third constant $\beta \in \mathbb{R}_{\geq 0}$, which we will need later to shift functions along the time axis. Altogether, we obtain the tradeoff function $g_e \colon \mathbb{R}_{>0} \to \mathbb{R}$ mapping driving time to energy consumption, which is defined as

$$g_e(x) := \frac{\alpha}{(x - \beta)^2} + \gamma. \tag{5.8}$$

For single edges, we always obtain $\beta = 0$ and assume driving time x to be *strictly* positive. Thus, the denominator $x - \beta$ is strictly positive and $g_e(x)$ is a finite real value. Furthermore, note that g_e is *decreasing* and *convex* on its domain $\mathbb{R}_{>0}$ in this case. Tradeoff functions of *paths* may require values $0 < \beta < x$ to reflect additional

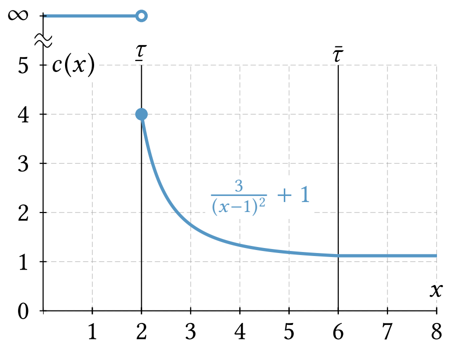


Figure 5.13: A consumption function, defined by a single tradeoff function with parameters $\alpha = 3$, $\beta = 1$, and $\gamma = 1$. The indicated subdomain borders induced by its minimum and maximum driving time, respectively, are $\underline{\tau} = 2$ and $\bar{\tau} = 6$.

time spent on previous edges. In the simplistic model discussed before, we have seen that tradeoff functions of paths may be piecewise linear. Similarly, we allow tradeoff functions in the realistic model to be defined as *piecewise* functions, so they may consist of multiple subfunctions of the form of Equation 5.8.

Given a tradeoff function $g \colon \mathbb{R}_{>0} \to \mathbb{R}$, we plug in the minimum and maximum driving time $\underline{\tau} \in \mathbb{R}_{>0}$ and $\bar{\tau} \in \mathbb{R}_{>0}$, respectively, to obtain the corresponding *consumption function* $c \colon \mathbb{R}_{\geq 0} \to \mathbb{R} \cup \{\infty\}$; see Figure 5.13 for an example. In general, we require consumption functions to be *continuous* in the interval $[\underline{\tau}, \infty)$, but not necessarily differentiable. In particular, we demand that $\beta < x$ holds for all subfunctions of the form of Equation 5.8 within their respective subdomain of $[\underline{\tau}, \bar{\tau}]$. Hence, the denominator $x - \beta$ is always strictly positive. Together with the assumption $\alpha \geq 0$, this implies that consumption functions are either *constant* (if $\alpha = 0$, in which case we further assume $\underline{\tau} = \bar{\tau}$) or *strictly decreasing* on the interval $[\underline{\tau}, \bar{\tau}]$, i. e., $c(x_1) < c(x_2)$ holds for all $x_1 \in [\underline{\tau}, \bar{\tau}]$ and $x_2 \in [\underline{\tau}, \bar{\tau}]$ with $x_1 > x_2$. At certain points below, we also make use of the *inverse function* $c^{-1} \colon [c(\bar{\tau}), c(\underline{\tau})] \to \mathbb{R}_{\geq 0}$ of a consumption function c. Observe that it is well-defined on the specified domain.

5.3.2 Linking Consumption Functions

If we want to generalize the BSP algorithm to propagation of consumption functions, we need operations that compute functions for (best) tradeoffs of *paths* instead of edges. In the previous section, we sketched how such consumption functions can be obtained for a simplistic model based on piecewise linear functions. Now, we describe how consumption functions are computed in the realistic model.

We define a *link operation* $\text{link} \colon \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ on the function space $\mathbb{F}$ of consumption functions as specified above. Given two consumption functions c_1 and c_2 representing energy consumption on two paths P_1 and P_2, respectively, linking c_1 and c_2 results in a consumption function $c := \text{link}(c_1, c_2)$ that maps driving time spent when traversing the path $P := P_1 \circ P_2$ to the *minimum* possible energy consumption (bar battery constraints). Let $\underline{\tau}_1 \in \mathbb{R}_{>0}$ and $\bar{\tau}_1 \in \mathbb{R}_{>0}$ denote the minimum and maximum driving

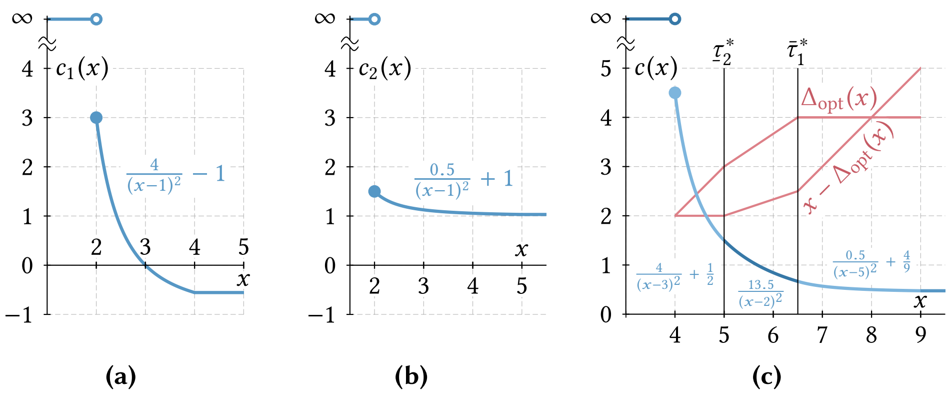


Figure 5.14: Linking two consumption functions. (a) The consumption function c_1, defined by the parameters $\alpha_1 = 4$, $\beta_1 = 1$, $\gamma_1 = -1$, $\underline{\tau}_1 = 2$, and $\bar{\tau}_1 = 4$. (b) The consumption function c_2, defined by the parameters $\alpha_2 = 0.5$, $\beta_2 = 1$, $\gamma_2 = 1$, $\underline{\tau}_2 = 2$, and $\bar{\tau}_2 = 5$. (c) The consumption function $c = \mathrm{link}(c_1, c_2)$, with $c(x) = c_1(\Delta_{\mathrm{opt}}(x)) + c_2(x - \Delta_{\mathrm{opt}}(x))$ on the interval $[4, 9]$. It is defined by three subfunctions with indicated subdomains $[4, 5]$, $[5, 6.5]$, and $[6.5, 9]$. The figure also shows the functions Δ_{opt} and $x - \Delta_{\mathrm{opt}}$ (red), indicating the share of c_1 and c_2.

time of c_1, respectively. Similarly, let $\underline{\tau}_2 \in \mathbb{R}_{>0}$ and $\bar{\tau}_2 \in \mathbb{R}_{>0}$ denote the corresponding driving times of c_2. Clearly, $c(x) = \infty$ holds for all $x < \underline{\tau}_1 + \underline{\tau}_2$ and $c(x) = c_1(\bar{\tau}_1) + c_2(\bar{\tau}_2)$ holds for all $x > \bar{\tau}_1 + \bar{\tau}_2$. For any remaining value $x \in [\underline{\tau}_1 + \underline{\tau}_2, \bar{\tau}_1 + \bar{\tau}_2]$, we have to determine times $x_1 \in [\underline{\tau}_1, \bar{\tau}_1]$ and $x_2 \in [\underline{\tau}_2, \bar{\tau}_2]$ on P_1 and P_2, respectively, that sum up to $x_1 + x_2 = x$ and minimize overall consumption (as in the simple model discussed in Section 5.3.1). We set $\Delta := x_1$ below, which yields

$$c(x) = \min_{\substack{\Delta \in [\underline{\tau}_1, \bar{\tau}_1] \\ \Delta \in [x - \bar{\tau}_2, x - \underline{\tau}_2]}} c_1(\Delta) + c_2(x - \Delta) \tag{5.9}$$

for all $x \in [\underline{\tau}_1 + \underline{\tau}_2, \bar{\tau}_1 + \bar{\tau}_2]$. In other words, to minimize the energy consumption for a given time $x \in [\underline{\tau}_1 + \underline{\tau}_2, \bar{\tau}_1 + \bar{\tau}_2]$, we have to divide the amount of time that exceeds the minimum possible total driving time $\underline{\tau}_1 + \underline{\tau}_2$ among the two paths, such that consumption is minimized. As in the simple example in Section 5.3.1, we would like to spend any additional time on the path corresponding to the function with steeper slope, since it provides the better tradeoff (we save more energy per additional unit of time spent on the corresponding edge). This is illustrated in Figure 5.14. In what follows, we formally derive the link operation and argue that the result is indeed a consumption function, i. e., a function that has the general form as in Equation 5.5, is continuous and decreasing on the interval $[\underline{\tau}_1 + \underline{\tau}_2, \infty)$, and whose tradeoff function is defined by subfunctions of the form as in Equation 5.8. Moreover, we show that the link operation can operate in linear time in the number of edges of P_1 and P_2.

Linking Functions Defined by Single Tradeoff Functions. For now, assume that each of the given functions c_1 and c_2 is defined by a *single* tradeoff subfunction, rather than multiple ones. For example, this is the case if both paths P_1 and P_2 consist of single edges, i. e., $P_1 = [u, v]$ and $P_2 = [v, w]$ for some $(u, v) \in E$ and $(v, w) \in E$. The result $c := \operatorname{link}(c_1, c_2)$ of linking c_1 and c_2 is a piecewise-defined consumption function, which may consist of multiple subfunctions of the form as in Equation 5.8; see Figure 5.14 for an example. Intuitively, the first subfunction of c represents a shifted part of the steeper input function for small values of x. It is followed by a combination of both functions and a subfunction that corresponds to the input function that is gentler for large values of x. Any of these parts may collapse (for example, the combined part only exists if one can pick admissible driving times such that both functions have identical negative slopes).

We now show how $c = \operatorname{link}(c_1, c_2)$ can be computed in constant time. Apparently, the best choice of Δ in Equation 5.9 depends on the value $x \in [\underline{\tau}_1 + \underline{\tau}_2, \bar{\tau}_1 + \bar{\tau}_2]$. Therefore, we consider the Δ-*function* $\Delta_{\text{opt}} \colon [\underline{\tau}_1 + \underline{\tau}_2, \bar{\tau}_1 + \bar{\tau}_2] \to \mathbb{R}_{\geq 0}$ that maps every admissible value of x to the optimal choice of Δ. Given this function, we immediately get for arbitrary $x \in \mathbb{R}_{\geq 0}$ that

$$c(x) = \begin{cases} \infty & \text{if } x < \underline{\tau}_1 + \underline{\tau}_2, \\ c_1(\bar{\tau}_1) + c_2(\bar{\tau}_2) & \text{if } x > \bar{\tau}_1 + \bar{\tau}_2, \\ c_1(\Delta_{\text{opt}}(x)) + c_2(x - \Delta_{\text{opt}}(x)) & \text{otherwise.} \end{cases} \tag{5.10}$$

Hence, we essentially need to compute Δ_{opt} to obtain the desired function c. To this end, consider an arbitrary fixed driving time $x \in [\underline{\tau}_1 + \underline{\tau}_2, \bar{\tau}_1 + \bar{\tau}_2]$. To identify the optimal value $\Delta_{\text{opt}}(x)$, we examine the derivative c'_x of the term $c_x(\Delta) := c_1(\Delta) + c_2(x - \Delta)$. It evaluates to

$$c'_x(\Delta) = \frac{2\alpha_1}{(\beta_1 - \Delta)^3} + \frac{2\alpha_2}{(x - \Delta - \beta_2)^3},$$

where α_1, β_1, α_2, and β_2 are the corresponding coefficients in the tradeoff functions of c_1 and c_2. We assume that $\alpha_1 > 0$ and $\alpha_2 > 0$ are positive (the other cases $\alpha_1 = 0$ or $\alpha_2 = 0$ are trivial). Then, we obtain a unique zero Δ^*_x for this derivative under the assumption that $\beta_1 < \Delta$ and $\Delta < x - \beta_2$. This holds true for valid choices of Δ, because $\Delta \geq \underline{\tau}_1 > \beta_1$ and $\Delta \leq x - \underline{\tau}_2 < x - \beta_2$ always holds; see Equation 5.9. Therefore, solving the term $c'_x(\Delta) = 0$ for Δ yields

$$\Delta^*_x = \frac{x - \beta_2 + \beta_1 \sqrt[3]{\frac{\alpha_2}{\alpha_1}}}{1 + \sqrt[3]{\frac{\alpha_2}{\alpha_1}}}.$$

The value Δ^*_x minimizes energy consumption for an unrestricted distribution of driving times which sum up to x. However, from Equation 5.9 we get the additional constraints

$\Delta_{\text{opt}}(x) \geq \max\{\underline{\tau}_1, x - \bar{\tau}_2\}$ and $\Delta_{\text{opt}}(x) \leq \min\{\bar{\tau}_1, x - \underline{\tau}_2\}$. Since Δ_x^* is the unique zero of c_x' in the open interval $(\beta_1, x - \beta_2)$, monotonicity of c_x in the intervals $(\beta_1, \Delta_x^*]$ and $[\Delta_x^*, x - \beta_2)$ follows. Thus, we get

$$\Delta_{\text{opt}}(x) = \begin{cases} \max\{\underline{\tau}_1, x - \bar{\tau}_2\} & \text{if } \Delta_x^* < \max\{\underline{\tau}_1, x - \bar{\tau}_2\}, \\ \min\{\bar{\tau}_1, x - \underline{\tau}_2\} & \text{if } \Delta_x^* > \min\{\bar{\tau}_1, x - \underline{\tau}_2\}, \\ \Delta_x^* & \text{otherwise.} \end{cases} \tag{5.11}$$

Equation 5.10 and Equation 5.11 together are sufficient to specify the desired function c. Since we want to explicitly represent c using tradeoff functions, we now derive the actual subfunctions that define c, depending on the value Δ_x^*.

First, solving the conditions $\Delta_x^* < \max\{\underline{\tau}_1, x - \bar{\tau}_2\}$ and $\Delta_x^* > \min\{\bar{\tau}_2, x - \underline{\tau}_2\}$ in Equation 5.11 for x yields four equivalencies in total, namely

$$\Delta_x^* < \underline{\tau}_1 \quad \Leftrightarrow \quad x < \underline{\tau}_1^* := \underline{\tau}_1 + \beta_2 + (\underline{\tau}_1 - \beta_1)\sqrt[3]{\frac{\alpha_2}{\alpha_1}}, \tag{5.12}$$

$$\Delta_x^* < x - \bar{\tau}_2 \quad \Leftrightarrow \quad x > \bar{\tau}_2^* := \bar{\tau}_2 + \beta_1 + (\bar{\tau}_2 - \beta_2)\sqrt[3]{\frac{\alpha_1}{\alpha_2}}, \tag{5.13}$$

$$\Delta_x^* > \bar{\tau}_1 \quad \Leftrightarrow \quad x > \bar{\tau}_1^* := \bar{\tau}_1 + \beta_2 + (\bar{\tau}_1 - \beta_1)\sqrt[3]{\frac{\alpha_2}{\alpha_1}}, \tag{5.14}$$

$$\Delta_x^* > x - \underline{\tau}_2 \quad \Leftrightarrow \quad x < \underline{\tau}_2^* := \underline{\tau}_2 + \beta_1 + (\underline{\tau}_2 - \beta_2)\sqrt[3]{\frac{\alpha_1}{\alpha_2}}. \tag{5.15}$$

Note that we obtain similar statements when solving for equality, e. g., we have $\Delta_x^* = \underline{\tau}_1$ if and only if $x = \underline{\tau}_1^*$. Consequently, we also get $\Delta_x^* > \underline{\tau}_1$ if and only if $x > \underline{\tau}_1^*$ (and analogous results for Equations 5.13–5.15). To obtain the actual function c and its subfunctions, we use the following Lemma 5.10.

Lemma 5.10. *Let c_1 and c_2 be two consumption functions, such that each is defined by a single tradeoff function g_1 and g_2, respectively. Moreover, let $\underline{\tau}_1$, $\bar{\tau}_1$, $\underline{\tau}_2$, and $\bar{\tau}_2$ denote their respective minimum and maximum driving times. Then the following statements hold for their derivatives g_1' and g_2' (even if we replace all occurrences of the relation "$\leq$" with "$=$" in the equivalencies below).*

1. $g_1'(\underline{\tau}_1) \leq g_2'(\underline{\tau}_2) \Leftrightarrow \underline{\tau}_1^* \leq \underline{\tau}_1 + \underline{\tau}_2 \leq \underline{\tau}_2^*$,
2. $g_1'(\bar{\tau}_1) \leq g_2'(\bar{\tau}_2) \Leftrightarrow \bar{\tau}_1^* \leq \bar{\tau}_1 + \bar{\tau}_2 \leq \bar{\tau}_2^*$,
3. $g_1'(\bar{\tau}_1) \leq g_2'(\underline{\tau}_2) \Leftrightarrow \bar{\tau}_1^* \leq \underline{\tau}_2^*$,
4. $g_1'(\underline{\tau}_1) \leq g_2'(\bar{\tau}_2) \Leftrightarrow \underline{\tau}_1^* \leq \bar{\tau}_2^*$.

Proof. All equivalencies follow after simple rearrangements. As an example, we show the first part of the first statement, namely, $g_1'(\underline{\tau}_1) \leq g_2'(\underline{\tau}_2)$ if and only if $\underline{\tau}_1^* \leq \underline{\tau}_1 + \underline{\tau}_2$. For $i \in \{1,2\}$, let α_i and β_i denote the coefficients of the tradeoff function g_i as in Equation 5.8. We exploit that $\alpha_1 > 0$, $\alpha_2 > 0$, $\underline{\tau}_1 > \beta_1$, and $\underline{\tau}_2 > \beta_2$ must hold to get

$$\begin{aligned}
& & g_1'(\underline{\tau}_1) &\leq g_2'(\underline{\tau}_2) \\
&\Leftrightarrow & -\frac{2\alpha_1}{(\underline{\tau}_1 - \beta_1)^3} &\leq -\frac{2\alpha_2}{(\underline{\tau}_2 - \beta_2)^3} \\
&\Leftrightarrow & (\underline{\tau}_2 - \beta_2)^3 &\geq \frac{2\alpha_2}{2\alpha_1}(\underline{\tau}_1 - \beta_1)^3 \\
&\Leftrightarrow & \underline{\tau}_2 &\geq \beta_2 + \sqrt[3]{\frac{\alpha_2}{\alpha_1}}(\underline{\tau}_1 - \beta_1) \\
&\Leftrightarrow & \underline{\tau}_1 + \underline{\tau}_2 &\geq \underline{\tau}_1 + \beta_2 + \sqrt[3]{\frac{\alpha_2}{\alpha_1}}(\underline{\tau}_1 - \beta_1) = \underline{\tau}_1^*.
\end{aligned}$$

All other statements follow from similar rearrangements. □

Together with Equations 5.12–5.15, Lemma 5.10 enables us to construct the desired function $c = \text{link}(c_1, c_2)$, depending on the slopes (i. e., the derivatives) of c_1 and c_2 at their respective subdomain borders. Exploiting that $\underline{\tau}_1 \leq \bar{\tau}_1$ and $\underline{\tau}_2 \leq \bar{\tau}_2$ hold by definition, we obtain the function c after the following case distinction. As in Lemma 5.10, let g_1 denote the (unique) tradeoff function defining c_1 and similarly, let g_2 be the tradeoff function defining c_2. We consider the slopes of these tradeoff functions at certain subdomain borders of c_1 and c_2. Without loss of generality, assume that $g_1'(\underline{\tau}_1) \leq g_2'(\underline{\tau}_2)$ holds (the other case is symmetric). This leaves us with only three possible cases, which are presented below.

1. $g_1'(\underline{\tau}_1) \leq g_2'(\underline{\tau}_2) \leq g_1'(\bar{\tau}_1) \leq g_2'(\bar{\tau}_2)$: Consider the relevant subdomain borders $\underline{\tau}_1 + \underline{\tau}_2$ and $\bar{\tau}_1 + \bar{\tau}_2$ corresponding to the minimum and maximum driving time of the function c obtained after linking c_1 and c_2. By the first, second, and third statement of Lemma 5.10, we know that

$$\underline{\tau}_1^* \leq \underline{\tau}_1 + \underline{\tau}_2 \leq \underline{\tau}_2^* \leq \bar{\tau}_1^* \leq \bar{\tau}_1 + \bar{\tau}_2 \leq \bar{\tau}_2^*.$$

For arbitrary values $x \in [\underline{\tau}_1 + \underline{\tau}_2, \underline{\tau}_2^*)$, we use the fact $x \geq \underline{\tau}_1^*$ and Equation 5.12 to infer the inequality $\Delta_x^* \geq \underline{\tau}_1$. Similarly, the fact $x < \bar{\tau}_2^*$ and Equation 5.13 yield $\Delta_x^* > x - \bar{\tau}_2$. Hence, we have $\Delta_x^* \geq \max\{\underline{\tau}_1, x - \bar{\tau}_2\}$ and the first case of Equation 5.11 does not apply. On the other hand, we know that both $x < \bar{\tau}_1^*$ and $x < \underline{\tau}_2^*$ hold, so by Equation 5.14 and Equation 5.15 we get $x - \underline{\tau}_2 < \Delta_x^* < \bar{\tau}_1$. This means that we are in the second case of Equation 5.11 and therefore, $\Delta_{\text{opt}}(x) = \min\{\bar{\tau}_1, x - \underline{\tau}_2\} = x - \underline{\tau}_2$ is the optimal choice for any $x \in [\underline{\tau}_1 + \underline{\tau}_2, \underline{\tau}_2^*)$. Making similar observations for the cases $x \in [\underline{\tau}_2^*, \bar{\tau}_1^*)$ and $x \in [\bar{\tau}_1^*, \bar{\tau}_1 + \bar{\tau}_2)$, we obtain corresponding values $\Delta_{\mathrm{opt}}(x) = \Delta_x^*$ and $\Delta_{\mathrm{opt}}(x) = \bar{\tau}_1$, respectively. This yields the desired function $c = \mathrm{link}(c_1, c_2)$, which is given by

$$c(x) = \begin{cases} \infty & \text{if } x < \underline{\tau}_1 + \underline{\tau}_2, \\ c_1(x - \underline{\tau}_2) + c_2(\underline{\tau}_2) & \text{if } \underline{\tau}_1 + \underline{\tau}_2 \le x < \underline{\tau}_2^*, \\ c_1(\Delta_x^*) + c_2(x - \Delta_x^*) & \text{if } \underline{\tau}_2^* \le x < \bar{\tau}_1^*, \\ c_1(\bar{\tau}_1) + c_2(x - \bar{\tau}_1) & \text{if } \bar{\tau}_1^* \le x < \bar{\tau}_1 + \bar{\tau}_2, \\ c_1(\bar{\tau}_1) + c_2(\bar{\tau}_2) & \text{otherwise.} \end{cases}$$

2. $g_1'(\underline{\tau}_1) \le g_1'(\bar{\tau}_1) \le g_2'(\underline{\tau}_2) \le g_2'(\bar{\tau}_2)$: In this case, we know by the four statements in Lemma 5.10 that the order $\underline{\tau}_1^* \le \bar{\tau}_1^* \le \underline{\tau}_1 + \underline{\tau}_2 \le \bar{\tau}_1 + \bar{\tau}_2 \le \underline{\tau}_2^* \le \bar{\tau}_2^*$ must hold. Equations 5.12–5.15 yield $\Delta_x^* \ge \max\{\underline{\tau}_1, x - \bar{\tau}_2\}$ and $\Delta_x^* > \min\{\bar{\tau}_1, x - \underline{\tau}_2\}$ in the whole subdomain $[\underline{\tau}_1 + \underline{\tau}_2, \bar{\tau}_1 + \bar{\tau}_2)$. Consequently, we obtain the optimal value $\Delta_{\mathrm{opt}}(x) = \min\{\bar{\tau}_1, x - \underline{\tau}_2\}$ and the function $c = \mathrm{link}(c_1, c_2)$ is defined as

$$c(x) = \begin{cases} \infty & \text{if } x < \underline{\tau}_1 + \underline{\tau}_2, \\ c_1(\bar{\tau}_1) + c_2(x - \bar{\tau}_1) & \text{if } \underline{\tau}_1 + \underline{\tau}_2 \le x < \bar{\tau}_1 + \underline{\tau}_2, \\ c_1(x - \underline{\tau}_2) + c_2(\underline{\tau}_2) & \text{if } \bar{\tau}_1 + \underline{\tau}_2 \le x < \bar{\tau}_1 + \bar{\tau}_2, \\ c_1(\bar{\tau}_1) + c_2(\bar{\tau}_2) & \text{otherwise.} \end{cases}$$

3. $g_1'(\underline{\tau}_1) \le g_2'(\underline{\tau}_2) \le g_2'(\bar{\tau}_2) \le g_1'(\bar{\tau}_1)$: Along the lines of the first case, Lemma 5.10 and Equations 5.12–5.15 yield that the function $c = \mathrm{link}(c_1, c_2)$ evaluates to

$$c(x) = \begin{cases} \infty & \text{if } x < \underline{\tau}_1 + \underline{\tau}_2, \\ c_1(x - \underline{\tau}_2) + c_2(\underline{\tau}_2) & \text{if } \underline{\tau}_1 + \underline{\tau}_2 \le x < \underline{\tau}_2^*, \\ c_1(\Delta_x^*) + c_2(x - \Delta_x^*) & \text{if } \underline{\tau}_2^* \le x < \bar{\tau}_2^*, \\ c_1(x - \bar{\tau}_2) + c_2(\bar{\tau}_2) & \text{if } \bar{\tau}_2^* \le x < \bar{\tau}_1 + \bar{\tau}_2, \\ c_1(\bar{\tau}_1) + c_2(\bar{\tau}_2) & \text{otherwise.} \end{cases}$$

Due to convexity of both g_1 and g_2, no other cases remain. Hence, the function c constructed above is defined by at most five subfunctions (two of which are constant). In each expression, we can expand the functions c_1 and c_2 to obtain a term that has the general form of a tradeoff function as in Equation 5.8. In particular, the denominator in the tradeoff functions of both c_1 and c_2 is (strictly) positive in all cases, i. e., we always have $x > \beta$. Moreover, it is easy to verify that c is continuous and decreasing on the interval $[\underline{\tau}_1 + \underline{\tau}_2, \infty)$, by inspecting the corresponding limits at the endpoints of each subdomain of c. In conclusion, the link operation requires constant time in the special case where both input functions are defined by a single tradeoff function. Degenerate cases are possible, too. For example, if c_1 or c_2 is constant, the link operation becomes a simple shift on the x-axis and y-axis. Lemma 5.11 summarizes our results.

Lemma 5.11. *Given two consumption functions c_1 and c_2, each defined by a single tradeoff subfunction, the link operation* $\mathrm{link}(c_1, c_2)$ *requires constant time and its result is a consumption function that is uniquely described by at most three tradeoff subfunctions.*

Linking General Consumption Functions. We tackle the general case, where we are given two consumption functions c_1 and c_2, each possibly consisting of *multiple* tradeoff subfunctions with the general form as in Equation 5.8, and want to compute the function $c := \mathrm{link}(c_1, c_2)$. Consider a tradeoff subfunction g of c_1 and its subdomain $[\underline{\tau}_g, \bar{\tau}_g)$. The subfunction g itself induces a consumption function c_g with minimum driving time $\underline{\tau}_g$ and maximum driving time $\bar{\tau}_g$; see Equation 5.5. By the fact that c_1 is decreasing and by construction of the consumption function c_g, we get $c_g(x) \geq c_1(x)$ for all $x \in \mathbb{R}_{\geq 0}$ and $c_g(x) = c_1(x)$ for all $x \in [\underline{\tau}_g, \bar{\tau}_g)$. Since the same argument can be made for subfunctions of c_2, it follows directly from Equation 5.9 that we obtain c after applying the constant-time link operation to all pairs of consumption functions induced by subfunctions of c_1 and c_2. The lower envelope of all resulting *candidate subfunctions* yields the desired function c.

Obviously, the function c is again a piecewise function. Moreover, as the lower envelope of functions that are decreasing on a common domain must be decreasing on this domain as well, the function c is decreasing. Finally, we claim that c is also continuous on the interval $[\underline{\tau}, \infty)$, where $\underline{\tau} \in \mathbb{R}_{>0}$ denotes the minimum driving time of c. By construction of c, a discontinuity in the interval $[\underline{\tau}, \infty)$ corresponds to a discontinuity of some candidate subfunction c^*. Further, we know that c^* has exactly one discontinuity at its minimum driving time $\underline{\tau}^* \in \mathbb{R}_{>0}$. By continuity of c_1 and c_2, there must be another candidate subfunction whose *maximum* driving time is $\underline{\tau}^*$ and whose function value coincides with c^* at this point, unless $\underline{\tau}^* = \underline{\tau}$. The claim follows and therefore, the function space of consumption functions is indeed closed under the link operation. Also, note that the link operation is commutative and associative.

The running time of the naïve link operation described above is quadratic in the number of subfunctions of c_1 and c_2. In what follows, we show how the complexity of the link operation for general consumption functions can be reduced to linear time. In our experiments, the number of subfunctions per consumption function was relatively small on average, so the speedup provided by the more sophisticated linear-time method was limited (less than 10 %). Hence, the algorithm described below may rather be considered to be of theoretical interest. It exploits the fact that consumption functions are *convex* in the general case as well, which we now prove formally.

In what follows, we say that a consumption function c with minimum driving time $\underline{\tau} \in \mathbb{R}_{>0}$ is *convex* if c convex on the interval $[\underline{\tau}, \infty)$. Note that this holds true for consumption functions of single edges; see Section 5.3.1. Consider the Δ-function Δ_{opt} of $c = \mathrm{link}(c_1, c_2)$, defined as the optimal choice of Δ in Equation 5.9. (We did not formally prove that the value Δ is distinct in the general case, but we may as well pick the *minimum* value Δ that fulfills Equation 5.9 to ensure that Δ_{opt} is well-defined.) Presuming that c_1 and c_2 are convex, the following Lemma 5.12 shows that both Δ_{opt} and the value $x - \Delta_{\text{opt}}(x)$ increase with respect to $x \in [\underline{\tau}_1 + \underline{\tau}_2, \bar{\tau}_1 + \bar{\tau}_2]$.

Lemma 5.12. *Let $\Delta_{\text{opt}} \colon [\underline{\tau}_1 + \underline{\tau}_2, \bar{\tau}_1 + \bar{\tau}_2] \to \mathbb{R}_{\geq 0}$ denote the Δ-function that corresponds to two convex consumption functions c_1 and c_2 with corresponding minimum and maximum driving times $\underline{\tau}_1$, $\bar{\tau}_1$, $\underline{\tau}_2$, and $\bar{\tau}_2$. Moreover, let $\bar{\Delta}_{\text{opt}}(x) := x - \Delta_{\text{opt}}(x)$ be defined on the domain of Δ_{opt}. Then both Δ_{opt} and $\bar{\Delta}_{\text{opt}}$ are continuous and increasing.*

Proof. We begin by showing that Δ_{opt} is increasing. Assume for contradiction that this is not the case, i. e., for some value x in the domain of Δ_{opt}, there are values $\varepsilon > 0$ and $\delta > 0$ such that $\Delta = \Delta_{\text{opt}}(x) > \Delta_{\text{opt}}(x + \varepsilon) = \Delta - \delta$. First of all, note that the inequality $c_1(\Delta) + c_2(x - \Delta) \leq c_1(\Delta - \delta) + c_2(x - \Delta + \delta)$ must hold, since Δ minimizes this term by definition of Δ_{opt}. Further, $\Delta = \Delta_{\text{opt}}(x)$ is the *smallest* among all values that minimize the term by definition, so plugging in $\Delta - \delta < \Delta$ actually yields a *strictly* greater result. Analogously, we have $c_1(\Delta - \delta) + c_2(x + \varepsilon - \Delta + \delta) \leq c_1(\Delta) + c_2(x + \varepsilon - \Delta)$, as this term is minimized by $\Delta - \delta$. Therefore, we obtain

$$\begin{aligned} c_1(\Delta - \delta) - c_1(\Delta) &\leq c_2(x + \varepsilon - \Delta) - c_2(x + \varepsilon - \Delta + \delta) \\ &\leq c_2(x - \Delta) - c_2(x - \Delta + \delta) \\ &< c_1(\Delta - \delta) - c_1(\Delta), \end{aligned}$$

which is a contradiction. Here, we exploit the fact that c_2 is convex and decreasing and hence, $c_2(x - \Delta) - c_2(x - \Delta + \delta)$ must be decreasing with respect to x for fixed values Δ and δ (the gap between two function values with constant difference on the x-axis must decrease if x increases).

Regarding $\bar{\Delta}_{\text{opt}}$, monotonicity follows from a very similar argument. As before, assume for contradiction that $\bar{\Delta}_{\text{opt}}(x) > \bar{\Delta}_{\text{opt}}(x + \varepsilon)$ for some $\varepsilon > 0$, so the inequality $x - \Delta_{\text{opt}}(x) > x + \varepsilon - \Delta_{\text{opt}}(x + \varepsilon)$ holds. We plug in $\Delta = \Delta_{\text{opt}}(x)$ and $\Delta + \delta = \Delta_{\text{opt}}(x + \varepsilon)$ to obtain $\delta > \varepsilon > 0$. As in the first case, we get $c_1(\Delta) + c_2(x - \Delta) \leq c_1(\Delta + \delta) + c_2(x - \Delta - \delta)$ and $c_1(\Delta + \delta) + c_2(x + \varepsilon - \Delta - \delta) < c_1(\Delta) + c_2(x + \varepsilon - \Delta)$ by the definition of Δ_{opt}. Along the lines of the first case, this yields a contradiction. Finally, the fact that both $\Delta_{\text{opt}}(x)$ and $x - \Delta_{\text{opt}}(x)$ increase with respect to x implies that $\Delta_{\text{opt}}(x)$ must be continuous. □

The following Lemma 5.13 proves that linking two convex consumption functions indeed yields a decreasing and convex function. Consequently, Lemma 5.12 applies to all consumption functions of the general form.

Lemma 5.13. *Given two consumption functions c_1 and c_2 that are convex on their subdomains $[\underline{\tau}_1, \bar{\tau}_1]$ and $[\underline{\tau}_2, \bar{\tau}_2]$ of admissible driving times, the function $c := \text{link}(c_1, c_2)$ is convex on the interval $[\underline{\tau}_1 + \underline{\tau}_2, \bar{\tau}_1 + \bar{\tau}_2)$.*

Proof. Assume for the sake of contradiction that $c = \operatorname{link}(c_1, c_2)$ is not convex on the indicated interval. We use the previous Lemma 5.12, which implies that the Δ-function with respect to c_1 and c_2 is increasing. Moreover, both (right) derivatives c_1' and c_2' are increasing on their respective subdomains $[\underline{\tau}_1, \bar{\tau}_1)$ and $[\underline{\tau}_2, \bar{\tau}_2)$ by assumption, as c_1 and c_2 are decreasing and convex. Given that c is not convex, its (right) derivative c' must be decreasing on some subinterval of $[\underline{\tau}_1 + \underline{\tau}_2, \bar{\tau}_1 + \bar{\tau}_2)$. Thus, there exist values $x \in [\underline{\tau}_1 + \underline{\tau}_2, \bar{\tau}_1 + \bar{\tau}_2)$ and $\varepsilon > 0$ such that $x + \varepsilon < \bar{\tau}_1 + \bar{\tau}_2$ and we get

$$\begin{aligned} c'(x) &> c'(x + \varepsilon) \\ &= c_1'(\Delta_{\text{opt}}(x + \varepsilon)) + c_2'(x + \varepsilon - \Delta_{\text{opt}}(x + \varepsilon)) \\ &\geq c_1'(\Delta_{\text{opt}}(x)) + c_2'(x - \Delta_{\text{opt}}(x)) \\ &= c'(x), \end{aligned}$$

which is a contradiction. This completes the proof. □

Given that the functions Δ_{opt} and $\bar{\Delta}_{\text{opt}}$ (as defined in Lemma 5.12) of an arbitrary pair of consumption functions are continuous and increasing, we are able to perform the link operation in a single coordinated linear scan, where we keep track of Δ_{opt} and $\bar{\Delta}_{\text{opt}}$. For two piecewise functions c_1 and c_2, let $g_1^1, \dots, g_1^k$ and $g_2^1, \dots, g_2^\ell$ denote their defining tradeoff functions, given in increasing order of their subdomains. For some subfunction g_i^j with $i \in \{1, 2\}$ and $j \in \{1, \dots, k\}$ or $j \in \{1, \dots, \ell\}$, respectively, we denote by $[\underline{\tau}_i^j, \bar{\tau}_i^j)$ its subdomain and by c_i^j its induced consumption function. The linear-time link operation proceeds as follows. First, it links the consumption functions c_1^1 and c_2^1 induced by the two tradeoff functions g_1^1 and g_2^1 with least admissible driving times. This results in a new convex consumption function $\operatorname{link}(c_1^1, c_2^1)$, which is defined by at most three tradeoff functions. Let Δ_{opt} and $\bar{\Delta}_{\text{opt}}$ be the Δ-functions associated with this link operation. We determine the next pair of consumption functions that are linked. To this end, we consider the points $x_1^1 := \Delta_{\text{opt}}^{-1}(\bar{\tau}_1^1)$ and $x_2^1 := \bar{\Delta}_{\text{opt}}^{-1}(\bar{\tau}_2^1)$ at which the induced consumption functions c_1^1 and c_2^1 reach their maximum driving time in the linked function. If $x_1^1 < x_2^1$, we continue with $\operatorname{link}(c_1^2, c_2^1)$, so Δ_{opt} can take values greater than $\bar{\tau}_1^1$. Similarly, if $x_1^1 > x_2^1$ holds, we compute $\operatorname{link}(c_1^1, c_2^2)$ next, so that $\bar{\Delta}_{\text{opt}}$ may exceed $\bar{\tau}_2^1$. Finally, in the special case $x_1^1 = x_2^1$ we proceed with $\operatorname{link}(c_1^2, c_2^2)$.

We continue this procedure until we reach the maximum driving time and link the consumption functions induced by g_1^k and g_2^ℓ. The lower envelope of the (linear number of) computed consumption functions yields the desired result $\operatorname{link}(c_1, c_2)$. Obviously, the running time of this procedure is in $O(k + \ell)$.

5.3.3 Basic Approach

In what follows, we describe our *tradeoff function propagating (TFP)* algorithm, which generalizes the (exponential-time) BSP algorithm for the basic problem given in Section 5.1. For a source $s \in V$, a target $t \in V$, and an initial SoC $b_s \in [0, M]$, it computes the fastest s–t path and optimal driving times such that battery constraints are respected. The basic idea of the algorithm is to propagate consumption functions (defined by sequences of tradeoff subfunctions) and apply battery constraints on-the-fly. Hence, it does not need to explicitly maintain bivariate SoC functions.

Figure 5.15 shows pseudocode of TFP. To keep the number of label comparisons small, we use the same method as in Section 5.2.2, where each vertex $v \in V$ maintains a set $L_{\text{set}}(v)$ and a heap $L_{\text{uns}}(v)$ containing settled (i. e., extracted) and unsettled labels, respectively. Each label is a (piecewise) consumption function, mapping the driving time on a certain s–v path to energy consumption. We say that a label c_1 *dominates* another label c_2 if $c_1(x) \leq c_2(x)$ holds for all $x \in \mathbb{R}_{\geq 0}$. Moreover, the *key* of a label c is defined as $\text{key}(c) := \underline{\tau}_c$, i. e., its minimum driving time. The corresponding maximum energy consumption $c(\underline{\tau}_c)$ is used to break ties. As in Section 5.2.2, we maintain the invariant that for each $v \in V$, $L_{\text{uns}}(v)$ is empty or the unsettled label in $L_{\text{uns}}(v)$ with *minimum* key is not dominated by any settled label in $L_{\text{set}}(v)$. New labels are pushed into $L_{\text{uns}}(v)$. Whenever the minimum element of $L_{\text{uns}}(v)$ changes (because an element is added or extracted), we check whether the new minimum element is dominated by some settled label in $L_{\text{set}}(v)$ and discard it in this case.

Label sets and the priority queue are initialized in lines 1–5 of the algorithm depicted in Figure 5.15. Initially, all label sets are empty, except for the constant function $c_s \equiv M - b_s$ at the source s. It is represented by the pair $(0, M - b_s)$. The source vertex is also added to the priority queue, which uses the minimum key among any unsettled labels of a vertex as its key. In each step of its main loop, the algorithm extracts some vertex $u \in V$ with minimum key from the priority queue and settles the corresponding label c_u, by taking it from the set $L_{\text{uns}}(u)$ and inserting it into $L_{\text{set}}(u)$ (lines 7–9 in Figure 5.15). Afterwards, the priority queue and the set $L_{\text{uns}}(u)$ are updated accordingly (lines 12–16). Finally, outgoing edges are scanned (lines 17–24). For every edge $(u,v) \in E$, the function $c := \text{link}(c_u, c_{(u,v)})$ is computed. Note that c may violate battery constraints, so we set $c(x) = \infty$ for all driving times $x \in \mathbb{R}_{\geq 0}$ with $c(x) > M$ in line 19 and $c(x) = 0$ for all $x \in \mathbb{R}_{\geq 0}$ with $c(x) < 0$ in line 20 of Figure 5.15. To identify such violations efficiently, we first check whether 0 or M are in the domain of the inverse c^{-1} of c. If this is the case, we set the minimum driving time of c to $c^{-1}(M)$ or the maximum driving time to $c^{-1}(0)$, respectively. If the resulting function yields finite consumption for some driving time, it is added to $L_{\text{uns}}(v)$ and the key of v in the priority queue is updated, if necessary. Note that the algorithm is *label setting*, i. e., labels extracted from the queue are never dominated later on. Thus, an optimal (constrained) path is found as soon as a label at t is extracted. The minimum driving time of this label is the optimal driving time; see line 11 of Figure 5.15.

Dominance Tests. To efficiently test whether a consumption function c_1 dominates another consumption function c_2, we inspect their respective subfunctions $g_1^1, \dots, g_1^k$

```
// initialize label sets
foreach $v \in V$ do
    $L_{\text{set}}(v) \longleftarrow \emptyset$
    $L_{\text{uns}}(v) \longleftarrow \emptyset$
$L_{\text{uns}}(s) \longleftarrow \{(0, M - b_s)\}$
$Q$.insert($s$,0)
// run main loop
while $Q$.isNotEmpty() do
    // extract next vertex
    $u \longleftarrow Q$.minElement()
    $c_u \longleftarrow L_{\text{uns}}(u)$.deleteMin()
    $L_{\text{set}}(u)$.insert($c_u$)
    if $u = t$ then
        return $\text{key}(c_u)$
    // update priority queue
    if $L_{\text{uns}}(u)$.isNotEmpty() then
        $c \longleftarrow L_{\text{uns}}(v)$.minElement()
        $Q$.update($u$,$\text{key}(c)$)
    else
        $Q$.deleteMin()
    // scan outgoing edges
    foreach $(u,v) \in E$ do
        $c \longleftarrow \text{link}(c_u, c_{(u,v)})$
        $\underline{\tau}_c \longleftarrow \min\{x \in \mathbb{R}_{\geq 0} \cup \{\infty\} \mid x = \infty \vee c(x) \leq M\}$
        $\bar{\tau}_c \longleftarrow \max\{x \in \mathbb{R}_{\geq 0} \cup \{\infty\} \mid x = \underline{\tau}_c \vee c(x) \geq 0\}$
        if $c \not\equiv \infty$ then
            $L_{\text{uns}}(v)$.insert($c$)
            if $c = L_{\text{uns}}(v)$.minElement() then
                $Q$.update($v$,$\text{key}(c)$)
```

Figure 5.15: Pseudocode of the TFP algorithm. It takes as input a graph $G = (V, E)$ with a function $C\colon E \to \mathbb{F}$ that assigns a consumption function c_e to every edge $e \in E$, as well as a source $s \in V$, a target $t \in V$, a battery capacity $M \in \mathbb{R}_{\geq 0}$, and an initial SoC $b_s \in [0, M]$. It computes the minimum driving time from s to t such that the SoC at t is nonnegative.

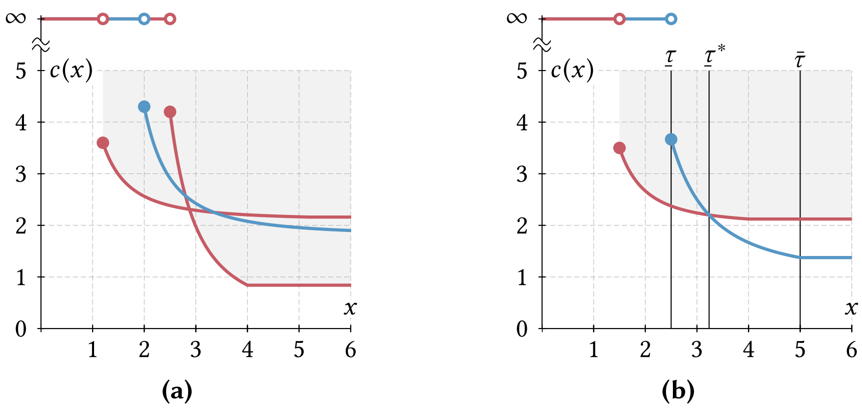


Figure 5.16: Dominance of consumption functions. The consumption function c (blue) of an unsettled label is compared to functions corresponding to a set of settled labels (red). Shaded areas indicate function values that are dominated by settled labels. (a) The consumption function c is not dominated by any label alone, but by the lower envelope of their two functions. In other words, for each $x \in \mathbb{R}_{\geq 0}$, one of the two settled labels yields lower consumption. Hence, the label c can be discarded. (b) The consumption function c is partially dominated by the settled label, so its minimum driving time can be increased from $\underline{\tau}$ to $\underline{\tau}^*$.

and $g_2^1, \ldots, g_2^\ell$. For some $i \in \{1, \ldots, k\}$ and $j \in \{1, \ldots, \ell\}$, consider the pair g_1^i and g_2^j of subfunctions (which have the form of Equation 5.8) and let $[\underline{\tau}_1^i, \bar{\tau}_1^i)$ and $[\underline{\tau}_2^j, \bar{\tau}_2^j)$ denote their respective subdomains. Observe that if the subdomains do not overlap, dominance can be checked easily by inspecting function values at the endpoints of these subdomains. Instead, let us assume now that their intersection is not empty, i. e., $[\underline{\tau}_1^i, \bar{\tau}_1^i) \cap [\underline{\tau}_2^j, \bar{\tau}_2^j) \neq \emptyset$. We can test in constant time whether $g_1^i(x) \leq g_2^j(x)$ holds for all $x \in [\max\{\underline{\tau}_1^i, \underline{\tau}_2^j\}, \min\{\bar{\tau}_1^i, \bar{\tau}_2^j\}]$ as follows. The subfunction g_1^i dominates the subfunction g_2^j in this interval if and only if $g_1^i(x) \leq g_2^j(x)$ holds for the two values $x = \max\{\underline{\tau}_1^i, \underline{\tau}_2^j\}$ and $x = \min\{\bar{\tau}_1^i, \bar{\tau}_2^j\}$ at the borders of their subdomain intersection, as well as the unique extreme point x of $g_1^i - g_2^j$ (if it falls within the considered intersection of their subdomains). Since we only have to compare subfunctions whose subdomains intersect, we can test whether c_1 dominates c_2 in a single linear scan (comparing subfunctions in increasing order of driving time). Hence, the running time of a dominance test is linear in the number of subfunctions of c_1 and c_2.

In our basic variant, the TFP algorithm implements dominance tests as follows. For a consumption function c with minimum driving time $\underline{\tau} \in \mathbb{R}_{>0}$ and maximum driving time $\bar{\tau} \in \mathbb{R}_{>0}$ in the set $L_{\text{uns}}(v)$ of some vertex $v \in V$, it performs a pairwise comparisons with each function in $L_{\text{set}}(v)$ to determine whether c is dominated by one of them. In doing so, the algorithm may miss that c is *partially* dominated or dominated only by a set of other labels; see Figure 5.16. In other words, even if c is not dominated by any settled label alone, it is possible that for some (or even all) admissible driving times $x \in [\underline{\tau}, \bar{\tau}]$ of c, there is a label $c^* \in L_{\text{set}}(v)$ with $c^*(x) \leq c(x)$. Although keeping c and eventually moving it to the set of settled labels does not affect correctness, it increases the label set size and may lead to unnecessary vertex scans. Instead of pairwise dominance checks, we may therefore try to identify dominated *parts* of the function c in question. We then compute a value $\underline{\tau}^* \in [\underline{\tau}, \bar{\tau}]$ such that for all $x \in [\underline{\tau}, \underline{\tau}^*]$, there is at least one function c^* contained in the set $L_{\text{set}}(v)$ with $c^*(x) \leq c(x)$. This value $\underline{\tau}^*$ can be computed in a single (coordinated) linear scan over the subfunctions of c and *all* subfunctions of labels in $L_{\text{set}}(v)$. If we obtain the value $\underline{\tau}^* = \bar{\tau}$ in this scan, c is dominated for all $x \in \mathbb{R}_{\geq 0}$, so we remove it from $L_{\text{uns}}(v)$. Otherwise, we set $\underline{\tau} = \underline{\tau}^*$. Analogously, we then compute a value $\bar{\tau}^* \in [\underline{\tau}^*, \bar{\tau}]$ such that c is dominated for all $x \in [\bar{\tau}^*, \bar{\tau}]$ and set $\bar{\tau} = \bar{\tau}^*$. Afterwards, we discard any subfunction of c whose subdomain does not intersect $[\underline{\tau}^*, \bar{\tau}^*]$.

Using the improved dominance check greatly reduces the number of labels in practice, at the cost of (little) additional overhead per dominance test. In particular, testing dominance now requires a coordinated scan over multiple subfunctions. We use the same test as described above to identify subfunctions that dominate a subfunction of c in the *whole* intersection of their respective subdomains. Note that we do not necessarily find the optimal values for $\underline{\tau}^*$ and $\bar{\tau}^*$ that way, and there may still exist values $x \in [\underline{\tau}^*, \bar{\tau}^*]$ with $c^*(x) \leq c(x)$ for some function $c^* \in L_{\text{set}}(v)$. However, we avoid expensive and error-prone intersection tests.

Path Retrieval. To obtain the actual s–t path, each label maintains two pointers to its corresponding parent label and vertex. To retrieve the optimal driving time (and speed) per edge, we explicitly store with each label c the function $\bar{\Delta}_{\text{opt}}$, given as $\bar{\Delta}_{\text{opt}}(x) = x - \Delta_{\text{opt}}(x)$, with respect to the previous link operation that lead to the creation of c. It is defined on the domain $[\underline{\tau}, \bar{\tau}]$ induced by the minimum driving time $\underline{\tau} \in \mathbb{R}_{>0}$ and the maximum driving time $\bar{\tau} \in \mathbb{R}_{>0}$ of c. Note that $\bar{\Delta}_{\text{opt}}$ is a piecewise function, with subfunctions that have the general form

$$\bar{\Delta}_{\text{opt}}(x) = x - \frac{x - \kappa}{\lambda} - \mu,$$

with nonnegative coefficients $\kappa \in \mathbb{R}_{\geq 0}$, $\lambda \in \mathbb{R}_{>0}$, and $\mu \in \mathbb{R}_{\geq 0}$; c. f. Equation 5.11 in Section 5.3.2. After the search has found the target t, the following backtracking routine yields the path itself and driving times on all edges. We start backtracking from t. Let x_t denote the minimum driving of the label c extracted at t and let $v \in V$ be the parent vertex of c. Then the driving time on the edge (v, t) is $\bar{\Delta}_{\text{opt}}(x_t)$, where $\bar{\Delta}_{\text{opt}}$ is the corresponding function stored with the label c. We continue this procedure recursively at its parent label with the driving time value $x_t - \bar{\Delta}_{\text{opt}}(x_t)$, until the source vertex s is reached.

A Polynomial-Time Heuristic. Even with improved dominance checks, TFP has exponential running time. However, the algorithm can easily be extended to a more efficient heuristic search, at the cost of inexact results. We propose a polynomial-time approach that is based on ε-dominance [Bat+11]. During the search, when performing the dominance test for a label $c \in L_{\text{uns}}(v)$ in the unsettled label set of some vertex $v \in V$, it is kept in $L_{\text{uns}}(v)$ only if it yields an improvement over settled labels in $L_{\text{set}}(v)$ by at least a certain fraction εM, with $\varepsilon \in (0,1]$, for some driving time. Thus, when identifying dominated parts of c, we test for each driving time $x \in \mathbb{R}_{\geq 0}$ whether $c^*(x) \leq c(x) + \varepsilon M$ holds for some settled label $c^* \in L_{\text{set}}(v)$. Observe that this implies that the number of settled labels per label set can become at most $\lceil 1/\varepsilon \rceil$. Given that the algorithm is label setting, each of these labels is extracted from the priority queue and added to $L_{\text{set}}(\cdot)$ at most once, so this yields polynomial running time in n and $\lceil 1/\varepsilon \rceil$.

Remarks. The heuristic variant of the TFP algorithm described above remains impractical for realistic ranges and reasonable values of ε, despite its polynomial running time. Therefore, we propose speedup techniques for TFP and its heuristic variant in the next sections, which are based on A* search and CH. Combining both techniques, we obtain our fastest algorithm, *CHAsp (CH, A*, Adaptive Speeds)*. The proposed speedup techniques do not alter the output of the algorithm, so correctness is maintained when using them with plain TFP. Although running time remains exponential in the worst case, we observe significant speedups in practice. Moreover, CHAsp can be combined with our polynomial-time heuristic in just the same way.

5.3.4 A* Search

A well-known approach to reduce (practical) running times in multicriteria scenarios is the adaptation of A* search [HNR68, MP10]. We propose two variants that compute a consistent potential function at query time, prior to running TFP. The potential of a vertex $v \in V$ is then added to the keys of all labels of v in TFP.

Potentials Based on Single-Criterion Search. We make use of two cost functions $\underaccent{\tilde}{d} \colon E \to \mathbb{R}_{\geq 0}$ with $\underaccent{\tilde}{d}(e) := \underaccent{\tilde}{\tau}_e$ for all $e \in E$ and $\underaccent{\tilde}{c} \colon E \to \mathbb{R}$ with $\underaccent{\tilde}{c}(e) := c_e(\bar{\tau}_e)$ for all $e \in E$, representing minimum driving time and minimum energy consumption on an edge, respectively. Similar to the approach by Tung and Chew [TC92], we aim at directing the search towards the target by preferring edges on (unrestricted) fastest paths. Before running TFP, we execute a single backward search (i. e., Dijkstra's algorithm running on the backward input graph $\bar{G}$) from the target $t \in V$, using the cost function $\underaccent{\tilde}{d}$. This yields, for each vertex $v \in V$, the distance $\text{dist}_{\underaccent{\tilde}{d}}(v,t)$, which is a lower bound on the (constrained) driving time from v to the t. We obtain a consistent potential function $\pi_\delta \colon V \to \mathbb{R}_{\geq 0}$ by setting $\pi_\delta(v) := \text{dist}_{\underaccent{\tilde}{d}}(v,t)$. By the triangle inequality, the potential function π_δ fulfills the condition $x - \pi_\delta(u) + \pi_\delta(v) \geq 0$ for all edges $(u,v) \in E$ and all *admissible* driving times $x \in [\underline{\tau}_e, \bar{\tau}_e]$. Thus, TFP is label setting when using π_δ, which implies that correctness of the algorithm is maintained.

To compute π_δ, the backward search described above visits all vertices in the (backward) graph. This can be wasteful, especially for small vehicle ranges. To avoid scans of unreachable vertices, we first run a backward search on $\bar{G}$ using the cost function $\underline{c}$ to compute, for all $v \in V$, lower bounds $\mathrm{dist}_{\underline{c}}(v,t)$ on the energy consumption required to reach the target t from v. This can be done by a label-correcting variant of Dijkstra's algorithm (as some edge costs may be negative); c. f. Section 5.2.4. We *prune* this search, i. e., we do not relax any outgoing edges, whenever the distance label of some scanned vertex exceeds the battery capacity M. Then, we run Dijkstra's algorithm on $\bar{G}$ to compute $\pi_\delta(v)$ as before, but restrict the search to vertices whose lower bound on energy consumption is below M. Afterwards, we restrict the TFP search to the same set of vertices.

Moreover, we use lower bounds $\mathrm{dist}_{\underline{c}}(\cdot,\cdot)$ for *pruning* in TFP: Before adding a new label c to the label set of some vertex $v \in V$, we first set $c(x) = \infty$ for all driving times $x \in \mathbb{R}_{\geq 0}$ with $c(x) + \mathrm{dist}_{\underline{c}}(v,t) > M$. To do so, we first check (in a linear scan over the subfunctions defining c^{-1}) whether $M - \mathrm{dist}_{\underline{c}}(v,t)$ is in the domain of the inverse function c^{-1} and set the minimum driving time of c to $c^{-1}(M - \mathrm{dist}_{\underline{c}}(v,t))$ if this is the case. Otherwise, we either obtain $c \equiv \infty$ or the function c remains unaffected.

Potentials Based on Bound Function Propagation. The potential function π_δ works well for common vehicle ranges, but may be too conservative if the consumption on the fastest path is very high. In such cases, it pays off to adapt the potential function $\pi_\varphi \colon V \times [0,M] \cup \{-\infty\} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ introduced in Section 5.2.4 to our scenario. Recall that the potential function π_φ incorporates the current SoC at some vertex to provide more accurate bounds. For each vertex $v \in V$, it uses a convex, piecewise linear function $\varphi_v \colon \mathbb{R} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ that maps the current SoC at v to a lower bound on the remaining driving time from v to the target $t \in V$. As higher SoC allows the EV to drive faster, φ_v is also *decreasing* with respect to SoC. Along the lines of Section 5.2.4, we say that the potential function π_φ is *consistent* if we obtain $x - \pi_\varphi(u,b) + \pi_\varphi(v, f_{(u,v)}(x,b)) \geq 0$ for all $(u,v) \in E$, $x \in [\underline{\tau}_{(u,v)}, \bar{\tau}_{(u,v)}]$, and $b \in [0,M]$. Here, $f_{(u,v)}$ denotes the SoC function of an edge $(u,v) \in E$ with minimum driving time $\underline{\tau}_{(u,v)}$ and maximum driving time $\bar{\tau}_{(u,v)}$, so the reduced cost must be nonnegative for any admissible parameter choice for $f_{(u,v)}$. We define $\pi_\varphi(u,-\infty) := \infty$. Together with the requirement that the potential $\pi_\varphi(t,b)$ at the target $t \in V$ equals 0 for all $b \in [0,M]$, consistent potentials maintain correctness of TFP; c. f. Section 5.2.4.

The functions representing the potentials $\pi_\varphi(\cdot,\cdot)$ are determined in a label-correcting backward search from t, similar to the search presented in Section 5.2.4. Figure 5.17 shows pseudocode of the search algorithm. It operates on the backward graph $\bar{G}$ of the input graph and maintains a piecewise linear function $\varphi_v \colon \mathbb{R} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ for

```
    // initialize labels
1   foreach $v \in V$ do
2       $\varphi_v \longleftarrow \emptyset$
3   $\varphi_t \longleftarrow [(0,0)]$
4   $Q$.insert($t$,0)
    // run main loop
5   while $Q$.isNotEmpty() do
6       $u \longleftarrow Q$.deleteMin()
7       foreach $(u,v) \in \bar{E}$ do
8           $\varphi_{(u,v)} \longleftarrow \mathrm{convert}(c_{(u,v)})$
9           $\varphi \longleftarrow \mathrm{link}(\varphi_u, \varphi_{(u,v)})$
10          if $\exists x \in \mathbb{R}\colon \varphi(x) < \varphi_v(x)$ then
11              $\varphi_v \longleftarrow \mathrm{merge}(\varphi_v, \varphi)$
12              $Q$.update($v$,key($\varphi_v$))
```

Figure 5.17: Pseudocode of the function propagating potential search for TFP. It requires an input graph $\bar{G} = (V, \bar{E})$ with a function $c\colon \bar{E} \to \mathbb{F}$ assigning consumption functions c_e to its edges $e \in \bar{E}$ and a target $t \in V$. For each $v \in V$, the algorithm computes a piecewise linear function $\varphi_v\colon \mathbb{R} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$, which maps SoC to a lower bound on driving time from v to t.

each $v \in V$, represented by a sequence $\Phi_v = [(b_1, x_1), \ldots, (b_k, x_k)]$ of breakpoints, such that $b_i < b_j$ for $i < j$, $\varphi_v(b) = \infty$ for $b < b_1$, and $\varphi_v(b) = x_k$ for $b \geq b_k$. We ignore battery constraints in the search, so we obtain lower bounds on SoC values, which can also become negative.

Each vertex stores a *single* label consisting of its (tentative) lower bound function. The search is initialized in lines 1–4 of Figure 5.17 with a function φ_t at the target $t \in V$ that evaluates to 0 for arbitrary nonnegative SoC, represented by the single breakpoint $(0,0)$. In the main loop, scanning an outgoing edge $(u,v) \in E$ of some vertex $u \in V$ corresponds to *linking* the two lower bounds representing the label at u and the edge (u,v). To this end, we first *convert* the consumption function $c_{(u,v)}$ mapping driving time to energy consumption to a piecewise linear function $\varphi_{(u,v)}$ mapping SoC to a lower bound on driving time (line 8 of Figure 5.17). Let $\underline{\tau} \in \mathbb{R}_{>0}$ and $\bar{\tau} \in \mathbb{R}_{>0}$ denote the minimum and maximum driving time of $c_{(u,v)}$, respectively. If $c_{(u,v)}$ is constant (i. e., $\underline{\tau} = \bar{\tau}$), we immediately obtain the lower bound defined by $\Phi_{(u,v)} := [(c_{(u,v)}(\underline{\tau}), \underline{\tau})]$. Otherwise, we consider two geometric lines ℓ_1 and ℓ_2 defined as follows; see also Figure 5.18. The first line ℓ_1 passes through the point $p = (c_{(u,v)}(\bar{\tau}), \bar{\tau})$ with slope equal to the (right) derivative of the inverse $c^{-1}_{(u,v)}$ of $c_{(u,v)}$ at $c_{(u,v)}(\bar{\tau})$. The second line ℓ_2 goes through $q = (c_{(u,v)}(\underline{\tau}), \underline{\tau})$ and its slope equals the (left) derivative of $c^{-1}_{(u,v)}$ at $c_{(u,v)}(\underline{\tau})$. Since $\underline{\tau} \neq \bar{\tau}$ and $c^{-1}_{(u,v)}$ is convex on its domain $[c_{(u,v)}(\bar{\tau}), c_{(u,v)}(\underline{\tau})]$, the lines ℓ_1 and ℓ_2 intersect in a unique point $r \in \mathbb{R}^2$. Then, a

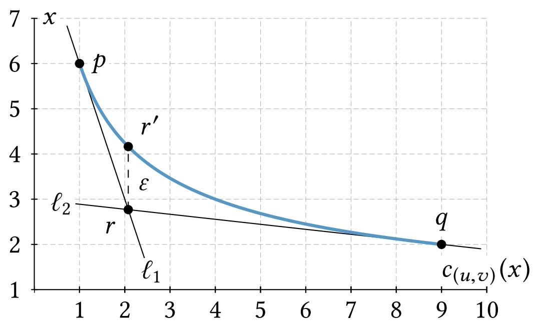


Figure 5.18: Constructing the lower bound function $\varphi_{(u,v)}$. It is defined by the endpoints p and q of the inverse of the consumption function $c_{(u,v)}$ with $\underline{\tau} = 2$ and $\bar{\tau} = 6$, and the intersection of the lines ℓ_1 and ℓ_2. The value ε indicates the maximum error of the lower bound. Recursion starts at r'.

convex lower bound $\varphi_{(u,v)}$ is defined by the breakpoints $[p,r,q]$. To increase accuracy of the lower bound, we may repeat this operation recursively for initial points p or q together with the unique point $r' \in \mathbb{R}^2$ on the inverse of $c_{(u,v)}$ that has the same x-coordinate as r; see Figure 5.18. Recursion stops as soon as the maximum difference between $c^{-1}_{(u,v)}$ and the function $\varphi_{(u,v)}$ induced by the current sequence of points falls below a predefined threshold $\varepsilon > 0$. Note that this difference is obtained in constant time, since it always occurs in the latest point r that was added to $\varphi_{(u,v)}$.

We link the function φ_u at u with the resulting lower bound $\varphi_{(u,v)}$ by computing, for any SoC value $b \in \mathbb{R}$, an optimal distribution of the available amount b of energy among the corresponding u–t path and the edge (u,v). Formally, the resulting function $\varphi = \text{link}(\varphi_u, \varphi_{(u,v)})$ evaluates to

$$\varphi(b) := \min_{b^* \in \mathbb{R}} \varphi_u(b^*) + \varphi_{(u,v)}(b - b^*)$$

for all $b \in \mathbb{R}$. This function can be computed in a linear scan over the breakpoints of both functions; see Section 5.2.6. To improve running times in practice, we can discard the label φ at this point if $\varphi(b) > M$ for all $b \in \mathbb{R}$.

After scanning the edge, we *merge* the resulting function φ with the function φ_v in the label of v, i. e., we compute the pointwise minimum $\varphi_v = \text{merge}(\varphi_v, \varphi)$ in a linear scan over the breakpoints of φ_v and φ. To ensure that the resulting function is again convex, we apply Graham's scan [Gra72] as in Section 5.2.4.

We define $\pi_\varphi(v,b) := \varphi_v(b)$ for all $v \in V$ and $b \in [0,M]$ once the search has terminated. The following Lemma 5.14 uses similar arguments as in the proof of Lemma 5.5 in Section 5.2.4 to prove that π_φ is indeed a consistent potential function.

Lemma 5.14. *The potential function π_φ is consistent.*

Proof. For some edge $(u,v) \in E$, let φ_u and φ_v be the piecewise linear functions at u and v, respectively, after the backward search has terminated. Let $\varphi_{(u,v)}$ denote the lower bound function of (u,v). We know that φ_u is upper bounded by the function computed by $\text{link}(\varphi_v, \varphi_{(u,v)})$, since the latter was merged into φ_u at some point during the backward search. For an arbitrary driving time $x \in \mathbb{R}_{\geq 0}$, let $b^* := c_{(u,v)}(x)$ denote the corresponding energy consumption on the edge (u,v). Then, we immediately obtain for all $b \in [0,M]$ that

$$\begin{aligned} x + \pi_\varphi(v, f_{(u,v)}(x,b)) &= x + \varphi_v(f_{(u,v)}(x,b)) \\ &\geq x + \varphi_v(b - c_{(u,v)}(x)) \\ &= c^{-1}_{(u,v)}(b^*) + \varphi_v(b - b^*) \\ &\geq \varphi_{(u,v)}(b^*) + \varphi_v(b - b^*) \\ &\geq \min_{b^* \in \mathbb{R}} \varphi_{(u,v)}(b^*) + \varphi_u(b - b^*) \\ &\geq \pi_\varphi(u,b), \end{aligned}$$

which implies that the reduced cost $x - \pi_\varphi(u,b) + \pi_\varphi(v, f_{(u,v)}(x,b))$ is nonnegative. This completes the proof. □

As soon as the backward search has terminated, we start the actual TFP search. During this search, we obtain the key of a label c at some vertex $v \in V$ by setting it to $\text{key}(c) := \min_{x \in \mathbb{R}_{\geq 0}} x + \pi_\varphi(v, M - c(x))$. Thus, labels closer to t (for the available SoC) get smaller keys. Computing the key requires a linear scan over the subfunctions defining c and the lower bound function φ_v at v. In each step of the scan, we update the minimum by evaluating the term $x + \varphi_v(M - c(x))$ at up to three values $x \in \mathbb{R}_{\geq 0}$, namely, the boundaries of the intersection of the subdomains of the considered subfunctions of c and φ_v and at the unique extreme point of their sum (if it falls within the current intersection of subdomains).

Implementation Details. When computing the potential function π_φ, the number of breakpoints of lower bounds can become quite large. Therefore, we reduce it as follows (while slightly deteriorating the quality of the bounds). Before applying Graham's scan, we replace consecutive pairs of breakpoints in the piecewise linear function with a single one if they are close to each other, i. e., their difference with respect to driving time or SoC is below a certain threshold $\Delta_x \in \mathbb{R}_{\geq 0}$ or $\Delta_b \in \mathbb{R}_{\geq 0}$, respectively. Two such points $p = (b_p, x_p)$ and $q = (b_q, x_q)$ are then replaced by the point $r := (\min\{b_p, b_q\}, \min\{x_p, x_q\})$. Furthermore, if two consecutive segments pq and qr with $p = (b_p, x_p)$, $q = (b_q, x_q)$, and $r = (b_r, x_r)$ have similar slopes $s_{pq} \approx s_{qr}$ (i. e., the difference $|s_{pq} - s_{qr}|$ is below some threshold $\Delta_s \in \mathbb{R}_{\geq 0}$), we replace them by a single segment from (b_p, x_p) to (b^*, x_r) with slope $\min\{s_{pq}, s_{qr}\}$, which uniquely defines the value $b^* \in \mathbb{R}$. Clearly, the modified function remains a lower bound. Moreover, consistency of the potential is maintained, as function values can only decrease and changes in the function are propagated by the search. Thus, all steps in the proof of Lemma 5.14 still apply.

Graham's scan and the breakpoint reduction step are performed on-the-fly during the merge operation. Moreover, we convert consumption functions c_e of all edges $e \in E$

to their corresponding lower bounds φ_e during preprocessing for faster query times. The thresholds ε to determine lower bound errors, Δ_x and Δ_b for close points, and Δ_s for similar slopes are tuning parameters. Smaller thresholds increase accuracy of bounds, but also slow down the backward search. Therefore, we set the above thresholds to $2^{\delta-\lfloor\log M\rfloor}$ in our experiments, where $\delta \in \mathbb{N}$ is a constant and M is the battery capacity (assumed to be given in kWh). Hence, bounds are more accurate for higher capacities (where the forward search becomes more expensive). The value of δ is again a tuning parameter. In our experiments, we use $\delta = 10$ for Δ_x (the resulting threshold is measured in seconds), $\delta = 17$ for Δ_b, and $\delta = 15$ for ε (both measured in Wh). For example, a battery capacity of 16 kWh yields $\Delta_x = 64$ (seconds), $\Delta_b = 2^{13}$ (Wh), and $\varepsilon = 2^{11}$ (Wh). The value $\Delta_s = 2^{-4}$ is constant and chosen independently of M (all parameters were determined in preliminary experiments).

5.3.5 Contraction Hierarchies and CHAsp

We propose an adaptation of CH to our scenario, which adds a preprocessing step to TFP for faster queries. As in plain CH [Gei+12b], vertices are contracted iteratively during preprocessing and *shortcut edges* are inserted to maintain distances. Similar to the approach from Section 5.2.5, we contract only a subset of the vertices, leaving an uncontracted *core* graph.

Since the SoC at a vertex is only known at query time in our setting, any shortcut has to store a *bivariate* SoC function. Figure 5.19 given further below shows an example of how the initial SoC influences energy consumption in our model. Their bivariate nature makes explicit construction and comparison of SoC functions rather involved. We say that an SoC function f_1 *dominates* another SoC function f_2 if $f_1(x,b) \geq f_2(x,b)$ holds for all $x \in \mathbb{R}_{\geq 0}$ and $b \in [0,M]$. Below, we examine simple representations of SoC functions in certain special cases that also enable efficient dominance tests. Afterwards, we describe how CH can utilize these simple SoC functions.

Simple SoC Functions. Consider the consumption functions $c_1,\ldots,c_{k-1}$ of the edges of a path $P = [v_1,\ldots,v_k]$ in G. Assume that $c_i(x) \geq 0$ holds for all driving times $x \in \mathbb{R}_{\geq 0}$ and $i \in \{1,\ldots,k-1\}$, i. e., all function values are nonnegative. In this case, battery constraints can render driving at high speed infeasible. On the other hand, recuperation never occurs on P. Therefore, the best speed on an edge depends solely on the slope of its consumption function, but not on the position of the edge in the path (in contrast to the situation discussed in Section 5.3.1 and sketched in Figure 5.11, where the order of edges clearly matters). Consequently, it is sufficient to first link the consumption functions and apply battery constraints only *once* afterwards. Lemma 5.15 proves this formally and specifies the resulting bivariate SoC function, which is represented by a single univariate (consumption) function.

Lemma 5.15. *Let $P = [v_1, \dots, v_k]$ be a path in G and let c_i denote the consumption function of the edge (v_i, v_{i+1}) for $i \in \{1, \dots, k-1\}$. If all consumption functions are nonnegative, i. e., $c_i(x) \geq 0$ holds for all $x \in \mathbb{R}_{\geq 0}$ and $i \in \{1, \dots, k-1\}$, the SoC function of P evaluates to*

$$f(x,b) = \begin{cases} -\infty & \textit{if } b < c(x), \\ b - c(x) & \textit{otherwise,} \end{cases}$$

where $c := \text{link}(\dots\text{link}(\dots\text{link}(c_1, c_2), \dots), c_{k-1})$ denotes the function obtained after iteratively linking the functions $c_1, \dots, c_{k-1}$.

Proof. First, consider the SoC function f_i of the consumption function c_i of an individual edge (v_i, v_{i+1}) with $i \in \{1, \dots, k-1\}$. It equals

$$f_i(x,b) = \begin{cases} -\infty & \text{if } b < c_i(x), \\ b - c_i(x) & \text{otherwise,} \end{cases}$$

since c_i has only nonnegative values by assumption, so the SoC at v_{i+1} never exceeds M. We prove the lemma by induction. Assume that for some $i \in \{1, \dots, k-2\}$, we are given the result $c_{1,\dots,i} := \text{link}(\dots\text{link}(\dots\text{link}(c_1, c_2), \dots), c_i)$ of linking all edges in $[v_1, \dots, v_{i+1}]$ and the corresponding SoC function is

$$f_{1,\dots,i}(x,b) = \begin{cases} -\infty & \text{if } b < c_{1,\dots,i}(x), \\ b - c_{1,\dots,i}(x) & \text{otherwise.} \end{cases}$$

We construct the SoC function $f_{1,\dots,i+1}$ using $c_{1,\dots,i+1} := \text{link}(c_{1,\dots,i}, c_{i+1})$. Since $c_{i+1}(x)$ is nonnegative for all $x \in \mathbb{R}_{\geq 0}$, we obtain $c_{1,\dots,i}(x) \leq c_{1,\dots,i+1}(x)$ for arbitrary driving times $x \in \mathbb{R}_{\geq 0}$. Hence, the path is infeasible for a pair of driving time $x \in \mathbb{R}_{\geq 0}$ and initial SoC $b \in [0, M]$ if and only if $b < \max\{c_{1,\dots,i}(x), c_{1,\dots,i+1}(x)\} = c_{1,\dots,i+1}(x)$. Otherwise, the function $c_{1,\dots,i+1}$ minimizes consumption on the path by definition of the link operation (still, no recuperation is possible). We obtain the function

$$f_{1,\dots,i+1}(x,b) = \begin{cases} -\infty & \text{if } b < c_{1,\dots,i+1}(x), \\ b - c_{1,\dots,i+1}(x) & \text{otherwise,} \end{cases}$$

which completes the proof. □

Note that the functions $c_1, \dots, c_{k-1}$ can actually be linked in arbitrary order, since the link operation is both commutative and associative. Thus, we can easily construct a shortcut for P by iteratively contracting its internal vertices $v_2, \dots, v_{k-1}$ in any given order, each time linking the consumption functions of both incident (shortcut) edges to compute a new shortcut. A symmetric argument holds for paths consisting only of nonpositive consumption functions.

Consequently, we allow contraction of a vertex if *all* (finite) values of consumption functions assigned to *any* of its incident edges have the same sign. Shortcuts inserted after contraction then have the simple form given in Lemma 5.15, so they can be computed efficiently. Furthermore, we can reuse dominance tests introduced in Section 5.3.3 to identify unnecessary shortcuts (we simply compare the two corresponding consumption functions). On real-world instances, where the majority of consumption values is positive, this approach already allows contraction of large parts of the graph (more than 50 % of the vertices in our tests). Nevertheless, the size of the resulting core graph is still too large to achieve significant speedups.

Discharging Paths. We discuss simple representations of SoC functions in more general cases, exploiting that most consumption values are positive in realistic instances. We say that a path P is *discharging* if the SoC on P never exceeds the (arbitrary) initial SoC, i. e., there exists no prefix of P that has negative consumption for any driving time. Subpaths with negative consumption are allowed, though. Note that it is not necessary to explicitly check whether the SoC exceeds M on a discharging path. We show how the SoC function of a discharging path can be represented by at most two consumption functions.

Clearly, a path consisting solely of edges with nonnegative consumption values (for arbitrary driving times) is discharging. We also showed how it can be represented by a single consumption function. As a more intricate example, assume we are given a path $P = P_1 \circ P_2$ consisting of two subpaths P_1 and P_2 that can be represented by two consumption functions c_1 and c_2. Let $\underline{\tau}_1$, $\overline{\tau}_1$, $\underline{\tau}_2$, and $\overline{\tau}_2$ denote the respective minimum and maximum driving times of c_1 and c_2. Moreover, assume that the consumption $c_1(x) > 0$ is *positive* for all $x \in \mathbb{R}_{\geq 0}$, while $c_2(x) \leq 0$ is *nonpositive* for all $x \in [\underline{\tau}_2, \infty)$. Finally, assume that $|c_1(\overline{\tau}_1)| \geq |c_2(\overline{\tau}_2)|$, i. e., the cost of P_1 is higher than the gain of P_2 for *any* driving time, so P is discharging. We derive the SoC function of P, represented by a *positive part* c^+ with $c^+(x) := c_1(x - \underline{\tau}_2)$ and a *negative part* $c^-(x)$ with $c^-(x) := c_2(x + \underline{\tau}_2)$. The original functions are shifted along the x-axis to simplify the analysis (note that the minimum driving time of c^- is 0). Given some initial SoC $b \in [0, M]$, the positive part c^+, and the negative part c^-, we define the *constrained positive part* $c_b^+ \colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ as the function that evaluates to

$$c_b^+(x) := \begin{cases} \infty & \text{if } b < c^+(x), \\ c^+(x) & \text{otherwise.} \end{cases}$$

The function applies battery constraints along P_1 for the initial SoC b; see Figure 5.19. Then, the SoC function f of the path P evaluates to $f(x, b) = b - \operatorname{link}(c_b^+, c^-)(x)$ for arbitrary $x \in \mathbb{R}_{\geq 0}$ and $b \in [0, M]$. Given some initial SoC, the function f first applies battery constraints on the positive part c^+ and links the resulting function c_b^+ with the negative part c^-. Since the underlying path P is discharging, we know that no further

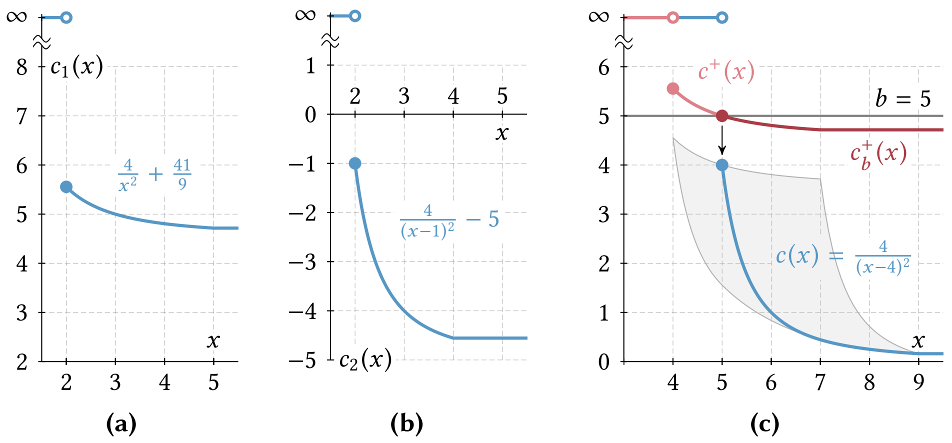


Figure 5.19: Construction of a consumption function depending on initial SoC. (a) The consumption function c_1 of a path P_1 with positive consumption. (b) The consumption function c_2 of a path P_2 with negative consumption. (c) Deriving the consumption function of the path $P := P_1 \circ P_2$. Due to battery constraints, the minimum driving time of c^+ is 5 for an initial SoC $b = 5$. This yields the consumption function $c := \text{link}(c_b^+, c^-)$ for the path P (the function c^-, which corresponds to c_2 being shifted to the left, is not shown). The shaded area indicates possible images of consumption functions for different values of initial SoC.

constraints need to be checked for c^-, so the function computed by $\text{link}(c_b^+, c^-)$ yields minimum energy consumption subject to driving time for the initial SoC b.

During preprocessing of CH, we iteratively construct new shortcuts from two existing (shortcut) edges in the current overlay graph. Hence, we have to be able to compute SoC functions representing general discharging paths from two given SoC functions of discharging paths. Assume we are given a discharging path P_1 whose SoC function is defined by two consumption functions c_1^+ and c_1^-, as described above. Similarly, we are given a discharging path P_2 with respective consumption functions c_2^+ and c_2^-. Observe that the path $P := P_1 \circ P_2$ must be discharging as well. We now construct the SoC function for P. Apparently, if we know the initial SoC, we can compute energy consumption on P by computing $\text{link}(\text{link}(\text{link}(c_1^+, c_1^-), c_2^+), c_2^-)$ and applying battery constraints *before* each link operation, like in the TFP algorithm (see Section 5.3.3). However, we want to represent P with only two consumption functions c^+ and c^-. Recall that the only constraint we have to check for discharging paths is whether the SoC drops below 0. We identify a new positive part c^+ as follows. Since both c_1^- and c_2^- are nonpositive for all admissible driving times, the constraint needs only to be checked for c_1^+ and c_2^+ (i. e., before the first and third link operation). To integrate these checks into a single positive part c^+, we first compute the consumption function $h := \text{link}(c_1^-, c_2^+)$. Clearly, the battery can only run empty on P_2 if this consumption function is *positive* for some admissible driving time (otherwise, we always gain more energy with c_1^- than is lost on c_2^+). To distinguish, we split h into a positive part $h^+ \colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ with $h^+(x) := \max\{h(x), 0\}$ and a negative part $h^- \colon \mathbb{R}_{\geq 0} \to \mathbb{R}_{\leq 0} \cup \{\infty\}$ with

$$h^-(x) := \begin{cases} \infty & \text{if } x < \bar{\tau} \text{ and } h(x) > 0, \\ \min\{h(x), 0\} & \text{otherwise,} \end{cases}$$

where $\bar{\tau} \in \mathbb{R}_{>0}$ denotes the maximum driving time of h. Since h is a valid consumption function, so are both h^+ and h^-. Observe that in case h yields nonnegative (nonpositive) energy consumption for all admissible driving times, the function h^- (h^+) always evaluates to 0 on its subdomain with finite image. We derive the positive part c^+ of P by setting $c^+(x) := \operatorname{link}(c_1^+, h^+)(x)$ for all $x \in \mathbb{R}_{\geq 0}$ and the negative part c^- of P by setting $c^-(x) := \operatorname{link}(h^-, c_2^-)(x + \underline{\tau})$ for all $x \in \mathbb{R}_{\geq 0}$, where $\underline{\tau} \in \mathbb{R}_{>0}$ is the minimum driving time of h^- (we shift the function to ensure that its minimum driving time is 0). The SoC function of P is obtained from c^+ and c^- as described above. Our way of splitting the function h ensures that battery constraints are only applied to prefixes of P with positive energy consumption.

Vertex Contraction. We use our insights about discharging paths to establish a preprocessing routine for CH. For the sake of simplicity, we assume that for each edge in the graph, the energy consumption is either nonnegative for all admissible driving times or nonpositive for all admissible driving times. Note that we can always enforce this by splitting an edge $e \in E$ with $c_e(\underline{\tau}_e) > 0$ and $c_e(\bar{\tau}_e) < 0$ into two consecutive edges corresponding to the positive part and the (shifted) negative part of c_e, respectively.

During preprocessing, we make sure that we only construct shortcuts with simple SoC functions. We say that a shortcut is *discharging* if it represents a discharging path. It is called *nonpositive* if it represents a path consisting solely of edges whose energy consumption is nonpositive for all admissible driving times. We only allow a vertex $v \in V$ to be contracted if all new shortcuts created as part of its contraction are discharging or nonpositive. We call v *active* in this case. Note that the number of active vertices grows as contraction proceeds, since this results in longer shortcuts, which are more likely to consist of significant positive parts. It remains to show how to decide efficiently whether a vertex is active during preprocessing and construct the necessary shortcuts if it is indeed contracted.

Assume that at some point during preprocessing, we want to contract a vertex $v \in V$ incident to two (shortcut) edges (u, v) and (v, w) in the current overlay graph. We have to determine whether a shortcut (u, w) can be constructed from (u, v) and (v, w) that is either discharging or nonpositive. Clearly, a nonpositive shortcut can be constructed if and only if both (u, v) and (v, w) are nonpositive. Otherwise, we want to know whether we can construct a discharging shortcut. Let P_1 be the underlying path in the original graph represented by (u,v) and let P_2 be the path represented by (v,w). (We get $P_1 = [u,v]$ if $(u,v) \in E$ is an original edge and $P_2 = [v,w]$ if $(v,w) \in E$ is an original edge.) We have to decide whether $P = P_1 \circ P_2$ is a discharging path, i. e., energy consumption is nonnegative on every prefix of P, regardless of the driving time. Apparently, this can only be the case if P_1 is a discharging path itself. For P_2, we distinguish two cases. First, if P_2 is discharging as well, so is P and the discharging shortcut (u,w) can be constructed as described above. Second, if P_2 is not a discharging path, it must consist solely of edges whose energy consumption is *nonpositive* for arbitrary (admissible) driving times, since a shortcut (v,w) would not have been created otherwise. Hence, P_2 is represented by a single consumption function c_2 with minimum and maximum driving time $\underline{\tau}_2 \in \mathbb{R}_{>0}$ and $\bar{\tau}_2 \in \mathbb{R}_{>0}$, respectively, such that $c_2(x) \leq 0$ for all $x \in [\underline{\tau}_2, \infty)$. To test whether a discharging shortcut (u,w) can be constructed in this case, consider the positive part c_1^+ and the negative part c_1^- of P_1 with corresponding maximum driving times $\bar{\tau}_1^+$ and $\bar{\tau}_1^-$. Then P is discharging if and only if $c_1^+(\bar{\tau}_1^+) + c_1^-(\bar{\tau}_1^-) + c_2(\bar{\tau}_2) \geq 0$, as these driving times minimize consumption on P (or any prefix of P that ends at a vertex of P_2). Moreover, we immediately obtain the positive part c^+ and the negative part c^- of the discharging path P, with $c^+(x) = c_1^+(x - \underline{\tau}_2)$ and $c^-(x) = \operatorname{link}(c_1^-, c_2)(x + \underline{\tau}_2)$ for all $x \in \mathbb{R}_{\geq 0}$.

Comparing Shortcut Candidates. In a bicriteria scenario, vertex contraction may result in multi-edges between the neighbors of a contracted vertex. In such cases, we only want to keep shortcuts whose SoC functions are not dominated by SoC functions of parallel edges. Hence, after the contraction of a vertex, we want to delete (parts of) SoC functions of shortcut candidates that are dominated by existing functions between the same pair of vertices (and vice versa). To this end, we require efficient dominance checks for SoC functions that are either *discharging*, i. e., represent a discharging path, or have nonpositive energy consumption for all admissible driving times.

Assume we are given two SoC functions f_1 and f_2 defined by the respective consumption functions c_1^+, c_1^-, c_2^+, and c_2^- (we assume that the positive part evaluates to 0 for all admissible driving times if consumption is always nonpositive). Further, assume that our goal is to remove dominated parts of f_2, i. e., we want to identify intervals $I \subseteq \mathbb{R}_{\geq 0}$ where $f_1(x,b) \geq f_2(x,b)$ holds for all driving times $x \in I$ and all values $b \in [0,M]$ of initial SoC. If both SoC functions have nonpositive consumption, each is represented by a single consumption function and we can immediately apply the dominance tests described in Section 5.3.3. For the remaining cases, note that a discharging SoC function cannot dominate a function with nonpositive consumption for all possible values of initial SoC. Thus, we consider the case where at least f_2 is discharging. We propose a method that may miss dominated parts of SoC functions, but requires only a linear scan over the consumption functions that define the SoC functions f_1 and f_2. Thereby, correctness is maintained, but unnecessary shortcuts may be inserted. Let $\varepsilon \geq 0$ denote the smallest nonnegative real value such that $c_1^-(x) \leq c_2^-(x) + \varepsilon$ holds for all $x \in \mathbb{R}_{\geq 0}$. This value can be computed in a linear scan over the subfunctions of c_1^- and c_2^-, similar to a dominance test. The following Lemma 5.16 shows that if $c_1^+(x) + \varepsilon \leq c_2^+(x)$ holds for some $x \in \mathbb{R}_{\geq 0}$, choosing a driving time of x for the positive part c_2^+ always results in a solution that is dominated by f_1, regardless of the initial SoC.

Lemma 5.16. *Given a nonpositive or discharging SoC function f_1 and a discharging SoC function f_2, such that their respective consumption functions are c_1^+, c_1^-, c_2^+, and c_2^-, let the value $\varepsilon \geq 0$ be defined as described above. If $c_1^+(x^+)+\varepsilon \leq c_2^+(x^+)$ holds for some $x^+ \in \mathbb{R}_{\geq 0}$, any solution where x^+ is the (optimal) amount of time spent for c_2^+ is dominated by f_1, i. e., we obtain either $c_2^+(x^+) = \infty$ or $f_1(x^+ + x^-, b) \geq f_2(x^+ + x^-, b) = b - (c_2^+(x^+) + c_2^-(x^-))$ for all $x^- \in \mathbb{R}_{\geq 0}$ and $b \in [0, M]$.*

Proof. Assume for the sake of contradiction that there exists some value $x^+ \in \mathbb{R}_{\geq 0}$ such that $c_1^+(x^+) + \varepsilon \leq c_2^+(x^+)$ and for some time $x^- \in \mathbb{R}_{\geq 0}$ and SoC $b \in [0, M]$, the value $b - (c_2^+(x^+) + c_2^-(x^-))$ is a feasible solution that is not dominated by $f_1(x^+ + x^-, b)$. Since $\varepsilon \geq 0$, we know that $c_1^+(x^+) \leq c_2^+(x^+)$ holds. This implies that $c_1^+(x^+) + c_1^-(x^-)$ is a feasible solution for an SoC of b (recall that the minimum driving time of c_1^- is 0). Finally, we know that $c_1^+(x^+) \leq c_2^+(x^+) - \varepsilon$ holds by assumption and $c_1^-(x^-) \leq c_2^-(x^-) + \varepsilon$ holds by the definition of ε. This yields

$$\begin{aligned} f_1(x^+ + x^-, b) &\geq b - \left(c_1^+(x^+) + c_1^-(x^-)\right) \\ &\geq b - \left(c_2^+(x^+) - \varepsilon + c_2^-(x^-) + \varepsilon\right), \end{aligned}$$

which contradicts our assumption and completes the proof. □

After creating a new shortcut (u, v) with positive part c^+, we compare it to existing shortcuts between $u \in V$ and $v \in V$ as follows. First, we compute the value ε defined above with respect to each existing shortcut. Then, we determine parts of c^+ that are dominated by existing positive parts (after we increase their consumption by ε) and set $c^+(x) = \infty$ for such values $x \in \mathbb{R}_{\geq 0}$. We do this in a coordinated linear scan over c^+ and the positive parts of consumption functions of all existing shortcuts, as in our basic dominance tests (see Section 5.3.3). If $c^+ \equiv \infty$ holds afterwards, we remove the shortcut. Analogously, we identify parts of SoC functions of each existing shortcut that are dominated by the SoC function of the new shortcut candidate.

Witness Search. Consider a discharging shortcut candidate (u, v) from $u \in V$ to $v \in V$ that is neither dominated by any existing parallel shortcut (from u to v) nor dominates an existing parallel shortcut itself. Before adding (u, v) to the graph, we run a witness search to test if the shortcut is necessary to maintain distances in the current overlay graph G' (for some values of driving times and SoC). An exact approach would

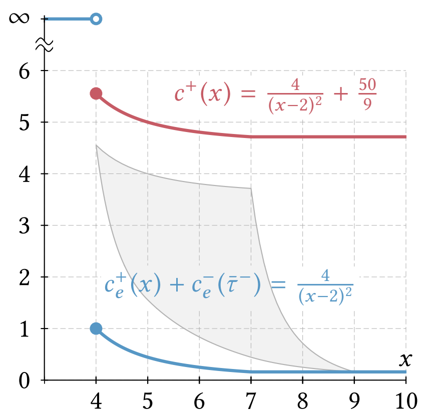


Figure 5.20: Bound functions computed by the witness search for the SoC function depicted in Figure 5.19c. The shaded area indicates possible values taken by this SoC function depending on driving time and SoC. It is upper bounded by the function c^+ (red) computed by the witness search. The lower bound (blue) is used to identify dominated parts of the SoC function if it represents a shortcut candidate.

compute bivariate SoC functions representing u–v paths in G' to identify dominated parts of the shortcut candidate. Like before, this is difficult and potentially expensive due to the lack of efficient operations for construction and comparison of such SoC functions. Instead, we only compute univariate *upper bounds* on energy consumption during witness search.

As a key idea, the search drops negative parts from SoC functions entirely, so labels in the search consist only of the positive parts. Clearly, these labels are upper bounds on energy consumption. Our witness search runs the basic TFP algorithm from u on G', but ignores battery constraints and links labels only with positive parts of overlay edges (as before, we assume that the positive part evaluates to 0 for all admissible driving times if consumption on the corresponding edge is always nonpositive). As a result, each label stores an upper bound on overall (finite) consumption that is independent of initial SoC and represented by a single consumption function c^+; see Figure 5.20. Moreover, since the majority of edges have positive consumption in realistic instances, the upper bounds have relatively small error in most cases.

Whenever a label c^+ at the head vertex v of the shortcut candidate $e = (u, v)$ is extracted during the witness search, we compare the label c^+ to the SoC function $f_{(u,v)}$ of the shortcut candidate as follows. Let c_e^+ and c_e^- denote the consumption functions defining $f_{(u,v)}$. Moreover, let $c_e^-(\bar{\tau}^-)$ be the *minimum consumption* of c_e^-. We know that c_e^+ is dominated for a driving time $x \in \mathbb{R}_{\geq 0}$ if $c^+(x) \leq c_e^+(x) + c_e^-(\bar{\tau}^-)$, i. e., the upper bound c^+ dominates a *lower bound* on the consumption of $f_{(u,v)}$; see Figure 5.20. We proceed along the lines of the dominance tests described in Section 5.3.3 to identify such values $x \in \mathbb{R}_{\geq 0}$ and remove them from the subdomain of admissible driving times of c_e^+. If $c_e^+ \equiv \infty$ holds afterwards, the shortcut is not required. Otherwise, the witness search stops once the minimum key of a scanned label exceeds the maximum driving time of the shortcut candidate. Note that during the search, we can discard labels if their minimum consumption exceeds the maximum consumption $c_e^+(\underline{\tau}^+) + c_e^-(\bar{\tau}^-)$ of the lower bound of the shortcut candidate.

Going even further, we replace upper bounds defined by *multiple* subfunctions that are computed during the witness search with *single* (less accurate) tradeoff functions. Then, witness search propagates labels of constant size, enabling faster operations and better locality. Moreover, we use lightweight dominance tests that employ pairwise label comparison (see Section 5.3.3), which are faster in this simplified scenario. Below, we describe how good bounds of constant descriptive complexity are computed.

Consider a piecewise-defined consumption function c with minimum and maximum driving time $\underline{\tau} \in \mathbb{R}_{>0}$ and $\bar{\tau} \in \mathbb{R}_{>0}$, respectively, that is defined by several tradeoff subfunctions $g_1, \ldots, g_k$. We seek a tradeoff function $\bar{g}$ that has the general form $\bar{g}(x) = \alpha/(x-\beta)^2 + \gamma$ (see Equation 5.8) for all $x \in [\underline{\tau}, \bar{\tau}]$ in the interval of admissible driving times. Further, we demand that $\bar{g}(x) \geq c(x)$ holds for all $x \in [\underline{\tau}, \bar{\tau}]$. To achieve this, we first set $\beta := \min_{i \in \{1,\ldots,k\}} \beta_i$, where β_i denotes the coefficient of the tradeoff subfunction g_i of c (see Equation 5.8). Then, we can fix the function values $\bar{g}(\underline{\tau}) := c(\underline{\tau})$ and $\bar{g}(\bar{\tau}) := c(\bar{\tau})$ at its domain borders to uniquely define the two remaining coefficients α and γ from Equation 5.8. In particular, we obtain

$$\gamma = \frac{c(\bar{\tau})(\beta - \bar{\tau})^2 - c(\underline{\tau})(\beta - \underline{\tau})^2}{(\beta - \bar{\tau})^2 - (\beta - \underline{\tau})^2} \qquad \text{and} \qquad \alpha = (c(\underline{\tau}) - \gamma)(\underline{\tau} - \beta)^2. \tag{5.16}$$

Lemma 5.17 shows that the resulting function is in fact an upper bound on c. Upper bounds are also *robust* towards incremental linking in the sense that the error does not increase if we recompute the upper bound whenever linking several bound functions results in a bound consisting of multiple subfunctions. This is due to the fact that the bounds are uniquely defined by their (minimum) coefficient β and the domain borders of the original functions, which remain unchanged in the upper bound.

During witness search, whenever linking two bound functions results in a function defined by more than one tradeoff subfunction, we compute and store the upper bound instead. Note that we do not even have to link functions explicitly, but simply compute the new coefficient β in a linear scan that simulates the link operation.

Lemma 5.17. *The function $\bar{g}$ defined above is an upper bound on the original consumption function c within the interval $[\underline{\tau}, \bar{\tau}]$, i. e., $\bar{g}(x) \geq c(x)$ holds for all $x \in [\underline{\tau}, \bar{\tau}]$.*

Proof. Let $g_1, \ldots, g_k$ denote the tradeoff subfunctions defining c and without loss of generality, assume that these subfunctions are given in increasing order of their admissible driving times. First, we argue that it is sufficient to prove the lemma for the case $k = 2$. To show this, we define an operation $\mathrm{bound}\colon \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ that takes as input two consumption functions, each defined by a *single* tradeoff subfunction, and computes an upper bound as described above. For $i \in \{1, \ldots, k-1\}$, consider two consumption functions c_i and c_{i+1} induced by two consecutive tradeoff functions g_i and g_{i+1} (see Section 5.3.2 for the definition of induced consumption functions). Let their corresponding minimum and maximum driving times be $\underline{\tau}_i$, $\bar{\tau}_i = \underline{\tau}_{i+1}$, and $\bar{\tau}_{i+1}$.

The bound operation computes the consumption function $c_{i,i+1} := \operatorname{bound}(c_i, c_{i+1})$ with minimum driving time $\underline{\tau}_i$, maximum driving time $\bar{\tau}_{i+1}$, and a single tradeoff subfunction $\bar{g}_{i,i+1}$. According to Equation 5.16, the coefficients of this tradeoff function depend only on the values $\beta = \min\{\beta_i, \beta_{i+1}\}$, the driving times $\underline{\tau}_i$ and $\bar{\tau}_{i+1}$, as well as the consumption values $c_i(\underline{\tau}_i) = \bar{g}_{i,i+1}(\underline{\tau}_i)$ and $c_{i+1}(\bar{\tau}_{i+1}) = \bar{g}_{i,i+1}(\bar{\tau}_{i+1})$ at the domain borders of $\bar{g}_{i,i+1}$. Linking the consumption function $c_{i,i+1}$ with another consecutive (induced) consumption function yields a new function defined by the same corresponding values. Consequently, the result $\operatorname{bound}(\dots\operatorname{bound}(\dots\operatorname{bound}(c_1, c_2), \dots), c_k)$ of iteratively applying the bound operation to the k induced consumption functions of c is the function that is defined by the coefficient $\beta = \min_{i \in \{0,\dots,k\}} \beta_i$, the minimum driving time $\underline{\tau} = \underline{\tau}_1$, the maximum driving time $\bar{\tau} = \bar{\tau}_k$, the maximum consumption $c(\underline{\tau}) = c_1(\underline{\tau}_1)$, and the minimum consumption $c(\bar{\tau}) = c_k(\bar{\tau}_k)$. This is exactly the function $\bar{g}$ defined above. Thus, we can construct $\bar{g}$ by iteratively applying the bound operation to consumption functions induced by the tradeoff subfunctions of c. To prove the lemma, we now show that each function constructed by the bound operation is in fact an upper bound on its *two* input functions. Observe that this implies that $\bar{g}$ is an upper bound on c within the interval $[\underline{\tau}, \bar{\tau}]$.

In the remainder of the proof, let c be a consumption function defined by two tradeoff subfunctions g_1 and g_2, which induce two consumption functions c_1 and c_2. We prove that the function $\bar{g} \colon \mathbb{R}_{>0} \to \mathbb{R}$ computed by $\operatorname{bound}(c_1, c_2)$ yields an upper bound on c on the interval $[\underline{\tau}, \bar{\tau}]$. Let the subdomains of g_1 and g_2 be $[\underline{\tau}, \tau)$ and $[\tau, \bar{\tau})$, respectively. By continuity of c on the interval $[\underline{\tau}, \bar{\tau}]$ and by continuity of both g_1 and g_2 on $\mathbb{R}_{>0}$, we know that $g_1(\tau) = g_2(\tau)$. To prove the lemma, we make use of the following three claims.

1. The inequality $\bar{g}(\tau) \geq g_1(\tau) = g_2(\tau)$ holds.
2. The slopes (i. e., the derivatives) of $\bar{g}$ and g_1 are equal at $\underline{\tau}$ if and only if $\bar{g} \equiv g_1 \equiv g_2$. Otherwise, the slope of $\bar{g}$ is *greater* at this point.
3. The slopes of $\bar{g}$ and g_2 are equal at $\bar{\tau}$ if and only if $\bar{g} \equiv g_1 \equiv g_2$. Otherwise, the slope of $\bar{g}$ is *smaller* at this point.

Then, $\bar{g}(\tau) \geq g_1(\tau)$ holds by our first claim, $\bar{g}$ and g_1 intersect at $\underline{\tau}$ by construction, and $\bar{g}(\underline{\tau} + \varepsilon) \geq g_1(\underline{\tau} + \varepsilon)$ holds for $\varepsilon > 0$ in the neighborhood of $\underline{\tau}$ by our second claim. This implies that $\bar{g}$ must be an upper bound on g_1 on the interval $[\underline{\tau}, \tau]$, because the functions $\bar{g}$ and g_1 can intersect at most twice in this interval unless $\bar{g} \equiv g_1$. This is easy to verify by computing the number of zeros of $\bar{g} - g_1$ within the considered interval $[\underline{\tau}, \tau]$. A similar argument holds for $\bar{g}$ and g_2 on the interval $[\tau, \bar{\tau}]$. Hence, the lemma follows after proving the three claims made above. We detail the rather technical proofs of these claims below.

Assume that the functions g_1 and g_2 are given as $g_i(x) = \alpha_i/(x - \beta_i)^2 + \gamma_i$ for all $x \in \mathbb{R}_{>0}$ and for $i \in \{1, 2\}$. For the sake of simplicity and without loss of generality, we presume that $\min\{\beta_1, \beta_2\} = 0$. Note that we can always enforce this property by *shifting* both functions (and their subdomains) along the x-axis. Afterwards, we obtain the same function $\bar{g}$ on the shifted subdomains. By a similar argument, we presume that $\gamma_1 = 0$ holds. Below, we consider the case $\beta_1 = 0$ and $\beta_2 \geq 0$. The case $\beta_1 \geq 0$ and $\beta_2 = 0$ is analogous. Since $\gamma_2 \in \mathbb{R}$ is allowed to become negative, no further case distinction is necessary.

To prove the first claim, we have to show that $\bar{g}(\tau) \geq g_1(\tau) = g_2(\tau)$ holds. For the sake of contradiction, assume $\bar{g}(\tau) < g_1(\tau)$. As mentioned before, continuity of the consumption function c on the interval $[\underline{\tau}, \bar{\tau}]$ implies that $g_1(\tau) = g_2(\tau)$, i. e.,

$$\frac{\alpha_1}{\tau^2} = \frac{\alpha_2}{(\tau - \beta_2)^2} + \gamma_2. \tag{5.17}$$

Furthermore, we know that c is a *convex* function (see Lemma 5.13), so when evaluating the derivatives of g_1 and g_2 at τ, we get the inequality

$$-\frac{2\alpha_1}{\tau^3} \leq -\frac{2\alpha_2}{(\tau - \beta_2)^3}. \tag{5.18}$$

Finally, we know that the inequalities $0 < \underline{\tau} < \tau < \bar{\tau}$, $\beta_2 \geq 0$, $\alpha_1 > 0$, $\alpha_2 > 0$, and $\tau > \beta_2$ hold by definition for consumption functions composed of multiple tradeoff subfunctions. We now show that altogether, these inequalities yield a contradiction. First, we plug the values of $\alpha \in \mathbb{R}_{\geq 0}$ and $\gamma \in \mathbb{R}$ from Equation 5.16 into $\bar{g}(\tau) = \alpha/\tau^2 + \gamma$. Afterwards, we replace $\gamma_2 = \alpha_1/\tau^2 - \alpha_2/(\tau - \beta_2)^2$ according to Equation 5.17 and exploit that the inequality $\alpha_1 \geq \alpha_2\tau^3/(\tau - \beta_2)^3 > 0$ holds by Equation 5.18 to obtain

$$\begin{aligned}
& \bar{g}(\tau) = \frac{g_1(\underline{\tau})\underline{\tau}^2(\bar{\tau}^2 - \tau^2) + g_2(\bar{\tau})\bar{\tau}^2(\tau^2 - \underline{\tau}^2)}{\tau^2(\bar{\tau}^2 - \underline{\tau}^2)} < g_1(\tau) \\
\Leftrightarrow \quad & g_1(\underline{\tau})\underline{\tau}^2(\bar{\tau}^2 - \tau^2) - g_1(\tau)\tau^2(\bar{\tau}^2 - \underline{\tau}^2) + g_2(\bar{\tau})\bar{\tau}^2(\tau^2 - \underline{\tau}^2) < 0 \\
\Leftrightarrow \quad & (\tau^2 - \underline{\tau}^2)\Big(\alpha_1(\bar{\tau}^2 - \tau^2)(\bar{\tau} - \beta_2)^2(\tau - \beta_2)^2 + \alpha_2\tau^2\bar{\tau}^2\big((\tau - \beta_2)^2 - (\bar{\tau} - \beta_2)^2\big)\Big) < 0 \\
\Rightarrow \quad & \frac{\alpha_2\tau^2}{\tau - \beta_2}\Big(\tau(\bar{\tau}^2 - \tau^2)(\bar{\tau} - \beta_2)^2 + \bar{\tau}^2(\tau - \beta_2)\big((\tau - \beta_2)^2 - (\bar{\tau} - \beta_2)^2\big)\Big) < 0 \\
\Leftrightarrow \quad & \beta_2(\bar{\tau}^2 - \tau^2)(2\bar{\tau}\tau - 2\bar{\tau}\beta_2 + \bar{\tau}^2 - \tau\beta_2) < 0 \\
\Leftrightarrow \quad & 2\bar{\tau}\tau - 2\bar{\tau}\beta_2 + \bar{\tau}^2 - \tau\beta_2 < 0.
\end{aligned}$$

This yields a contradiction, because we know that $0 \leq \beta_2 < \tau < \bar{\tau}$ holds. Thus, both $2\bar{\tau}\tau - 2\bar{\tau}\beta_2$ and $\bar{\tau}^2 - \tau\beta_2$ are positive terms and their sum cannot be negative.

For the second claim, we examine the slopes of g_1 and $\bar{g}$ at the domain border $\underline{\tau}$. Let the parameter $\alpha \in \mathbb{R}_{\geq 0}$ of $\bar{g}$ be defined as in Equation 5.16. Plugging in the coefficient $\beta = 0$ and the value of $\gamma \in \mathbb{R}$ according to Equation 5.16, we obtain

$$\alpha = \frac{\big(g_1(\underline{\tau}) - g_2(\bar{\tau})\big)\underline{\tau}^2\bar{\tau}^2}{\bar{\tau}^2 - \underline{\tau}^2}.$$

As before, we use $\gamma_2 = \alpha_1/\tau^2 - \alpha_2/(\tau - \beta_2)^2$ and the inequality $\alpha_1 \geq \alpha_2\tau^3/(\tau - \beta_2)^3 > 0$. For the difference between the derivatives $\bar{g}'$ and g_1' at $\underline{\tau}$, this yields

$$\begin{aligned}
\bar{g}'(\underline{\tau}) - g_1'(\underline{\tau}) &= \frac{2\alpha_1}{\underline{\tau}^3} - \frac{2\alpha}{\underline{\tau}^3} \\
&= \frac{2\left(g_2(\bar{\tau})\bar{\tau}^2\underline{\tau}^2 - \alpha_1\underline{\tau}^2\right)}{\underline{\tau}^3\left(\bar{\tau}^2 - \underline{\tau}^2\right)} \\
&= \frac{2\underline{\tau}^2}{\underline{\tau}^3\left(\bar{\tau}^2 - \underline{\tau}^2\right)}\left(\alpha_2\bar{\tau}^2\frac{(\tau - \beta_2)^2 - (\bar{\tau} - \beta_2)^2}{(\bar{\tau} - \beta_2)^2(\tau - \beta_2)^2} + \alpha_1\left(\frac{\bar{\tau}^2}{\tau^2} - 1\right)\right) \\
&\geq \frac{2\alpha_2\underline{\tau}^2}{\underline{\tau}^3(\bar{\tau}^2 - \underline{\tau}^2)}\left(\bar{\tau}^2\frac{(\tau - \beta_2)^2 - (\bar{\tau} - \beta_2)^2}{(\bar{\tau} - \beta_2)^2(\tau - \beta_2)^2} + \frac{\tau^3}{(\tau - \beta_2)^3}\left(\frac{\bar{\tau}^2}{\tau^2} - 1\right)\right) \\
&= \frac{2\alpha_2\beta_2\underline{\tau}^2(\bar{\tau} - \tau)^2(2\bar{\tau}\tau - 2\bar{\tau}\beta_2 + \bar{\tau}^2 - \tau\beta_2)}{\bar{\tau}^3(\bar{\tau}^2 - \underline{\tau}^2)(\bar{\tau} - \beta_2)^2(\tau - \beta_2)^3}.
\end{aligned}$$

As in the proof of the first claim, we observe that each term in the product of the numerator is nonnegative, while each term in the product of the denominator is positive. Moreover, the numerator is equal to 0 if and only if $\beta_2 = 0$ holds. Using Equation 5.17, it is easy to verify that this implies $\alpha_1 = \alpha_2$ and $\gamma_2 = 0$, which corresponds to the case where the three functions $\bar{g}$, g_1, and g_2 are equivalent.

Finally, we deal with the slopes of g_2 and $\bar{g}$ at $\bar{\tau}$ to prove the third claim. Below, we first replace the values α, α_2, and γ_2 as in our proof of the second claim. Afterwards, we exploit the fact that $(\tau^3 - x)(\bar{\tau} - \beta_2)^3 - (\bar{\tau}^3 - x)(\tau - \beta_2)^3$ decreases with increasing $x \in \mathbb{R}_{\geq 0}$, since its derivative with respect to x is $(\tau - \beta_2)^3 - (\bar{\tau} - \beta_2)^3 < 0$. After some further rearrangements, we obtain

$$\begin{aligned}
g_2'(\bar{\tau}) - \bar{g}'(\bar{\tau}) &= \frac{2\alpha}{\bar{\tau}^3} - \frac{2\alpha_2}{(\bar{\tau} - \beta_2)^3} \\
&\geq \frac{2\alpha_2\left((\tau^3 - \underline{\tau}^2\beta_2)(\bar{\tau} - \beta_2)^3 - (\bar{\tau}^3 - \underline{\tau}^2\beta_2)(\tau - \beta_2)^3\right)}{\bar{\tau}(\bar{\tau}^2 - \underline{\tau}^2)(\bar{\tau} - \beta_2)^3(\tau - \beta_2)^3} \\
&\geq \frac{2\alpha_2\left((\tau^3 - \tau^2\beta_2)(\bar{\tau} - \beta_2)^3 - (\bar{\tau}^3 - \tau^2\beta_2)(\tau - \beta_2)^3\right)}{\bar{\tau}(\bar{\tau}^2 - \underline{\tau}^2)(\bar{\tau} - \beta_2)^3(\tau - \beta_2)^3} \\
&= \frac{2\alpha_2\beta_2(\bar{\tau} - \tau)^2(\bar{\tau}\tau - \bar{\tau}\beta_2 + \bar{\tau}\tau - \tau\beta_2 + \tau^2 - \tau\beta_2)}{\bar{\tau}(\bar{\tau}^2 - \underline{\tau}^2)(\bar{\tau} - \beta_2)^3(\tau - \beta_2)^2}.
\end{aligned}$$

Again, we end up with products for which all factors are nonnegative (and strictly positive in case of the denominator). As before, the numerator equals 0 if and only if $\bar{g} \equiv g_1 \equiv g_2$. Hence, all three claims hold and the proof is complete. □

CHAsp Queries. To answer queries, plain CH use a bidirectional search, which scans only upward edges in the input graph enriched with shortcuts obtained during preprocessing. In our case, however, the SoC at the target vertex $t \in V$ is not known at query time. This makes backward search difficult, since it would require us to propagate bivariate SoC functions. Instead, we first run (at query time) a BFS from t on the backward graph (including shortcuts from preprocessing), scanning and marking only edges to vertices of higher rank. Afterwards, we execute the TFP algorithm, starting from the source vertex $s \in V$ and scanning upward edges, core edges, and marked downward edges in the graph. For faster queries, we enhance TFP with one of the variants of A* search described in Section 5.3.4. We only compute potentials for vertices contained in the search graph of the query. We refer to the combination of CH and A* search for TFP as *CHAsp (CH, A*, Adaptive Speeds)*.

Implementation Details. During preprocessing, we determine the next vertex to be contracted using the measures Edge Difference (ED) and Cost of Queries (CQ) according to Geisberger et al. [Gei+12b]. To reflect the complexity of SoC functions, we add another term *Shortcut Complexity (SC)*, which is defined as $|c^+| + k|c^-|$ for the SoC function of a given shortcut candidate, where $|c^+|$ and $|c^-|$ denote the number of tradeoff subfunctions that define the positive and negative part of a shortcut, respectively, and $k \in \mathbb{N}$ is a tuning parameter. Using penalized weights for negative parts, we favor earlier contraction of SoC functions without a negative part (we use $k = 4$ in our experiments). The priority of a vertex is then set to $64\,\mathrm{ED} + \mathrm{CQ} + \mathrm{SC}$.

To reduce the running time of witness searches, we also employ a *settled node limit* [Gei+12b] of 128, which limits the maximum number of queue extractions per witness search. If multiple shortcut candidates with the same tail vertex $u \in V$ are constructed during contraction of a vertex, we save time by running only a single multi-target witness search from u. Finally, to improve performance of the backward searches during a query (BFS and potential computation), we explicitly construct and store their more lightweight search graphs from the input graph (enriched with shortcuts, but storing less complex cost functions) during preprocessing.

5.4 Experiments

We evaluate the algorithms for both problem settings examined in this chapter on our main test instance, Eur-PTV, and its subnetwork, Ger-PTV. Unless mentioned otherwise, we derive energy consumption from the PHEM model of a Peugeot iOn. All experiments reported in this section were conducted on machine-s. For details on input data and the machine specification, see Section 3.4. Given that most algorithms considered in this section have exponential running time in the worst case, we aborted queries if no solution was found after an hour of computation time in all experiments. Below, we present our results regarding algorithms for routes with charging stops (Section 5.4.1) and adaptive speeds (Section 5.4.2) in turn.

Table 5.1: Impact of core size on performance (CHArge, Ger-PTV, 16 kWh). We stopped contraction if the average degree in the core graph exceeded a certain threshold (Ø Deg.). We report the core size (# Vertices), preprocessing time, and average query times of 1 000 random queries answered by CHArge-H_ω, using BSS and mixed charging stations, respectively.

Core size		Prepr.	Query [ms]	
Ø Deg.	# Vertices	[h:m:s]	Only BSS	Mixed CS
8	344 066 (7.33%)	2:58	1 474.1	47 979.9
16	116 917 (2.49%)	4:01	536.5	1 669.0
32	65 375 (1.39%)	5:03	436.1	1 356.8
64	43 036 (0.91%)	7:07	449.8	1 408.8
128	30 526 (0.65%)	11:16	509.6	1 585.4
256	22 592 (0.48%)	20:22	647.5	2 098.5
512	17 431 (0.37%)	37:11	880.7	2 739.9
1 024	13 942 (0.29%)	1:05:51	1 264.6	3 934.2
2 048	11 542 (0.24%)	2:00:27	1 822.6	5 670.1

5.4.1 Charging Stops

To evaluate our algorithms CFP and CHArge, which integrate charging stops, we use locations of charging stations extracted from ChargeMap (see Section 3.4). We collected 13 810 charging stations for Eur-PTV and 1 966 (a subset) for Ger-PTV. We construct different charging functions to model certain types of stations, namely, battery swapping stations (BSS), superchargers (charging an empty battery to 50 % SoC in 20 minutes and a maximum of 80 % in 40 minutes), as well as regular stations with fast (44 kW), medium (22 kW), or slow (11 kW) charging. For the three latter types of functions, we use the physical model of Uhrig et al. [Uhr+15] and approximate the corresponding charging functions with a piecewise linear function (using six breakpoints at 0 %, 80 %, 85 %, 90 %, 95 %, and 100 % SoC). We set initialization time to three minutes for BSS and one minute for all other types of charging stations. If not stated otherwise, queries are always generated by picking source and target vertices uniformly at random and an initial SoC of $b_s = M$

Evaluating Queries. We discuss preprocessing and query performance of our algorithms. We only report the fastest exact method (CHArge with the potential π_φ) and our heuristic approaches—for results on plain CFP see below. We consider two different scenarios. In the first, all charging stations are BSS, whereas the second uses a mixed composition of chargers (randomly picking 10 % of all stations as BSS, 20 % as superchargers, 30 % as fast chargers, and 40 % as slow chargers). The latter composition is fixed, i. e., all queries are run for the same assignment of charging station types. We use ChargeMap locations and the Peugeot iOn model.

Table 5.2: Preprocessing and query performance for different instances, charging station types (CS), and battery capacities (M). We report regular preprocessing times for CHArge (which also applies to the heuristics CHArge-H_φ and CHArge-H_ω) and preprocessing times for CHArge-$\mathrm{H}^\mathrm{A}_\omega$, the percentage of feasible queries, as well as exact and heuristic query times.

			Pr. [m:s]			Query [ms]			
Ins.	CS	M	Reg.	$\mathrm{H}^\mathrm{A}_\omega$	Feas.	Exact	H_φ	H_ω	$\mathrm{H}^\mathrm{A}_\omega$
Ger-PTV	BSS	16 kWh	5:03	4:33	100 %	1 398.0	994.5	436.1	20.9
	Mix.	16 kWh	5:03	4:33	100 %	8 628.7	1 495.2	1 356.8	155.2
	BSS	85 kWh	4:59	5:31	100 %	1 012.9	974.9	47.8	28.2
	Mix.	85 kWh	4:59	5:31	100 %	2 614.3	1 894.1	342.9	34.1
Eur-PTV	BSS	16 kWh	30:32	28:38	63 %	10 785.8	7 566.7	9943.3	207.4
	Mix.	16 kWh	30:32	28:38	63 %	24 147.6	10 039.3	17 630.3	2 694.0
	BSS	85 kWh	30:16	29:47	100 %	47 921.0	35 060.4	1 021.7	41.1
	Mix.	85 kWh	30:16	29:47	100 %	86 192.5	48 243.2	26 866.8	599.6

Table 5.1 shows details on preprocessing effort and query performance for different core sizes on Ger-PTV (for a battery capacity of 16 kWh). In this experiment, vertex contraction during preprocessing was stopped as soon as the average vertex degree in the core graph reached the threshold shown in the first column of the table. We report resulting core graph sizes, preprocessing times, and query times of CHArge-H_ω (the fastest query variant that uses the same core as CHArge) for the BSS and mixed station composition, respectively.

Apparently, preprocessing effort increases with decreasing core size. We achieve best query performance at an average core degree of 32. At higher degrees, the rather dense core causes query times to increase. Therefore, we use a core degree of 32 as threshold to stop vertex contraction in all further experiments. This results in relative core sizes of 1.3–1.7 % on Ger-PTV and Eur-PTV. Regarding query times, we observe that the mixed composition is harder to solve, because vertex potentials are less accurate in this case.

Table 5.2 shows detailed timings on performance for 1 000 queries on each considered instance. We evaluate two scenarios (only BSS and mixed composition of charging stations) on our instances Ger-PTV and Eur-PTV, for typical battery capacities (16 kWh and 85 kWh). Preprocessing times are quite practical, considering the problem at hand, ranging from about 5–30 minutes. As before, the mixed scenario is harder to solve. On the other hand, increasing the maximum battery capacity leads to faster queries. This can be explained by the fact that less charging is required, so goal direction is more helpful. A notable exception is the capacity of 16 kWh on Eur-PTV. In this setting, the number of feasible queries drops significantly, due to a highly non-uniform distribution of charging stations (sparse in parts of Southern and Eastern Europe; see Figure 3.10b

Table 5.3: Detailed query performance of CHArge (Ger-PTV, 85 kWh) for mixed and realistic charging stations (CS). For exact CHArge (Ex.) and the different heuristics (H_φ, H_ω, and H^A_ω), we report the number of settled labels (# Labels) and pairwise dominance checks (# Dom.) during the forward search, and the average running times. For the resulting trips, we report the percentage of feasible and optimal paths, as well as average and maximum increase in trip time compared to an optimal solution.

		Query			Result Quality			
CS	Algo.	# Labels	# Dom.	Time [ms]	Feas.	Opt.	Avg.	Max.
Mixed	Ex.	482 712	36 527 376	2 614.3	100 %	100 %	1.0000	1.0000
	H_φ	443 134	139 897	1 894.1	100 %	85 %	1.0010	1.0725
	H_ω	190 955	5 578 309	341.9	100 %	80 %	1.0004	1.0213
	H^A_ω	11 309	29 695	34.1	100 %	52 %	1.0200	1.2387
Realistic	Ex.	395 841	48 611 726	2 457.0	100 %	100 %	1.0000	1.0000
	H_φ	359 150	117 083	1 542.0	100 %	82 %	1.0007	1.0323
	H_ω	169 618	3 680 130	245.9	100 %	70 %	1.0009	1.0481
	H^A_ω	12 330	26 435	34.1	100 %	61 %	1.0128	1.1299

in Section 3.4). This benefits approaches based on the potential function π_φ, as we can often detect infeasibility already during potential computation (the lower bound on trip time evaluates to ∞). On the other hand, infeasible queries deteriorate the performance in many cases when using the potential function π_ω: Because the target is never reached, large parts of the graph are explored until the queue runs empty.

All in all, running times of the exact algorithm are below 10 seconds on average on Ger-PTV and below 90 seconds on Eur-PTV, which is quite notable given that we could not even run a single long-distance CFP query on this instance in several hours. Note that the mixed composition is a rather difficult configuration for our algorithms (we also tested other configurations, e. g., without BSS). When using the potential function π_ω for CHArge (not reported in the table), running times increase by up to an order of magnitude, depending on the scenario. Hence, plugging in the more sophisticated potential function π_φ pays off. For heuristic approaches, we see that—in contrast to CHArge—those based on the potential function π_ω are faster (except for instances with many infeasible queries).

Table 5.3 reports detailed figures on Ger-PTV (for the same set of queries as in Table 5.2), using a battery capacity of 85 kWh. We argue that this results in the most sensible queries: Due to a reasonably dense distribution of charging stations in Germany (see Figure 3.10b in Section 3.4), all queries are feasible and the target is always reachable with a small number of charging stops. Consequently, we obtain an average trip time of 3 h 26 min and an average charging time of two minutes on Ger-PTV for the mixed composition of charging stations. In contrast to that, harder

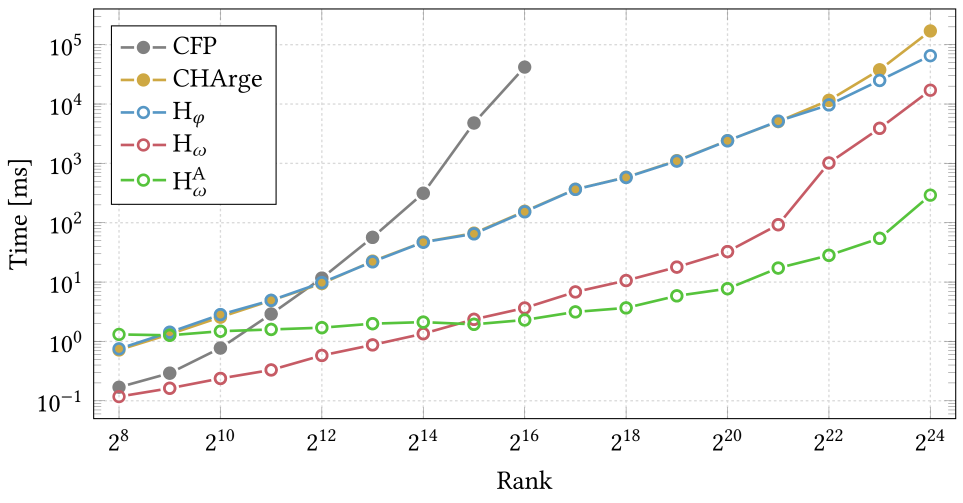


Figure 5.21: Running times of CHArge subject to Dijkstra rank. Each point in the plot corresponds to the median running time of 100 random queries on Eur-PTV with a 85 kWh battery and the mixed composition of charging stations.

queries across Europe on Eur-PTV (which we analyze rather to show scalability of our algorithms) yield an average trip time of over ten hours. We also add figures for a (currently) more realistic scenario, containing no BSS, 20 % superchargers, and 40 % of each fast and medium regular stations (which yields an average trip time of 3 h 39 min and an average charging time of 11 minutes). We see that the number of settled labels is a good indicator of the running time. Moreover, all approaches are quite practical. Computing optimal routes takes only a few seconds. The heuristic CHArge-H_ω provides a good trade-off between running times (some 300 ms) and resulting errors (below 0.1 % on average, below 5 % in the worst case). Our aggressive approach CHArge-H^A_ω allows query times of 34 ms on average, which is fast enough even for interactive applications. However, we see that solution quality is up to 24 % off the optimum in the worst case. Still, the average error of all heuristics is very low, and the optimal solution is found in more than half of the cases.

Evaluating Scalability. Figure 5.21 shows median running times on our hardest instance (Eur-PTV with mixed charging stations) distributed by their Dijkstra rank, which equals the number of queue extractions when running Dijkstra's algorithm from the source to the target with unconstrained driving time as edge costs [Bas+16, SS05]. We ran 100 queries per rank. Query times for CFP are only reported up to a rank of 2^{16}, because for higher ranks at least one query did not terminate within the predefined limit of one hour. Given a relatively large battery capacity of 85 kWh, charging stops are only necessary for queries of the highest Dijkstra ranks, starting

Table 5.4: Comparison of CHArge with existing work [Sto12a]. For different distributions of charging stations (CS) and ranges (M), we report preprocessing times of the existing approach [Sto12a], CHArge, and the heuristic CHArge-H_{ω}^{A} on Sgr-OSM. Regarding queries, we show the percentage of feasible paths, as well as exact and heuristic query times. Results from our competitor [Sto12a] are reported as is from a Core i3-2310M, 2.1 GHz.

		Prepr. [m:s]				Query [ms]				
CS	M	[Sto12a]	CHArge	H_{ω}^{A}	Feas.	[Sto12a]	CHArge	H_{φ}	H_{ω}	H_{ω}^{A}
r1000	100 km	51:41	11:37	2:30	100 %	539.0	122.1	93.7	38.1	2.7
r100	150 km	16:21	11:10	2:15	99 %	1 150.0	206.0	116.5	26.9	1.4
c643	100 km	—	11:21	2:32	98 %	—	326.3	282.6	48.3	3.3
c643	150 km	—	11:28	2:29	99 %	—	308.3	270.0	22.7	2.9

from (roughly) 2^{22}. Consequently, running times of our faster approaches increase significantly at this rank. It also becomes evident that all approaches have exponential running times, with timings of CHArge in the order of minutes for the highest ranks. Nevertheless, performance remains practical for ranks beyond 2^{20}. Running times of our fastest heuristic (CHArge-H_{ω}^{A}) are well below a second for the highest ranks.

Comparison with Related Work. We also consider the largest instance used to evaluate the fastest approach for routes with charging stops in the literature [Sto12a], kindly provided by the author. It is based on OSM data of Southern Germany (see the instance Sgr-OSM in Table 3.1 in Section 3.4) and elevation data from SRTM. Energy consumption is computed from the basic physical consumption model given by Equation 3.1 in Section 3.4. All charging stations in the original work [Sto12a] are BSS, picked uniformly at random from all vertices of the input graph. Table 5.4 shows detailed figures, using 100–1 000 randomly placed charging stations (r100, r1000) and battery capacities that translate to a certain range in the physical model. Query times are average values of 1 000 random queries. For completeness, we also run tests on ChargeMap stations (only BSS), 643 of which are included in the considered road network. As before, query times are average values of 1 000 random queries. We observe that our approach is faster with respect to both preprocessing and queries, even when taking differences in hardware into account. At the same time, CHArge is more general and not inherently restricted to BSS. Furthermore, note that a denser distribution of random charging stations enables faster query times: Although preprocessing effort increases slightly (more charging stations are kept in the core), the potential functions more often rightly assume that a station will be available close to the remaining path. We see that using charging station locations from ChargeMap (c643) results in a slightly harder instance. In conclusion, we are able to solve instances that are harder than those evaluated in the literature.

Table 5.5: Benefits of our approach (Eur-PG, 2 kWh). For TFP and TFP-dom. (improved dominance tests), we report the number of settled labels (# Labels), number of label comparisons (# Dom.), average and maximum running times, and relative driving time savings over the constrained paths found by the BSP algorithm on discretized speeds.

	Query				Path Savings	
Algo.	# Labels	# Dom.	Avg. [ms]	Max. [ms]	Avg. [%]	Max. [%]
BSP	30 990 276	21 300 657 522	47 755	779 756	—	—
TFP	103 119	4 399 002	444	14 347	2.7 %	9.4 %
TFP-dom.	46 228	700 546	103	3 851	2.7 %	9.4 %

5.4.2 Adaptive Speeds

To enable adaptive speeds on our input instances, we require a function of the form given in Equation 5.7 in Section 5.3.1 for each road segment, together with a reasonable interval of admissible driving speeds. We derived functions from two consumption models based on PHEM. The first is the same vehicle model as in the previous section, which is calibrated to a Peugeot iOn. The second is an artificial model [Tie+12] that, in contrast to the first, takes power demand of auxiliary consumers (e. g., air conditioning) into account. We extracted functions following Equation 5.7 from given samples of speed and energy consumption via regression. Combining reasonable minimum speeds for different road types (e. g., 80 km/h on motorways and 30 km/h in residential areas) with the posted speed limits (if higher), we get intervals of admissible speeds per road segment. As a result, 25 % and 38 % of the edges are nonconstant for the network of Germany and Europe, respectively. We denote the resulting instances by Ger-PG and Eur-PG when using the consumption model based on a Peugeot iOn. Similarly, Ger-AX and Eur-AX denote the respective instances based on the artificial model that takes auxiliary consumers into account. The amount of edges with negative consumption (for at least some travel times) is 7.8 % on Ger-AX, 12.2 % on Ger-PG, 9.6 % on Eur-AX, and 12.9 % on Eur-PG.

Unless mentioned otherwise, our study evaluates random *in-range* queries, i. e., we pick a source vertex $s \in V$ uniformly at random. Among all vertices in range from s with an initial SoC $b_s = M$, we pick the target $t \in V$ uniformly at random (as in Section 4.5.1). Since unreachable targets can be detected by backward search phases of A* search or by any algorithm for computing energy-optimal routes (see Chapter 4), this results in more difficult and interesting queries (recall that we do not consider charging stops in this section).

Model Validation. We have argued that an approach based fully on consumption *functions* unlocks both better tractability and improved solution quality compared

Table 5.6: Impact of core size on performance (CHAsp, Ger-PG, 16 kWh). Vertex contraction stopped when the average degree of active vertices in the core reached a given threshold (Ø Deg.). We report the resulting core size (# Vertices), preprocessing time, and average query times for 1 000 queries using CHAsp with potential functions π_δ and π_φ, respectively.

Core size		Prepr.	Query [ms]	
Ø Deg.	# Vertices	[h:m:s]	CHAsp-π_δ	CHAsp-π_φ
0	—	—	3 326.0	4 861.5
8	720 514 (15.36 %)	5:07	737.2	798.3
16	400 174 (8.53 %)	13:25	496.2	485.0
24	333 819 (7.11 %)	22:28	456.2	442.6
32	305 301 (6.51 %)	31:44	451.8	434.0
48	279 943 (5.97 %)	51:06	475.1	451.0
64	268 436 (5.72 %)	1:11:13	505.5	473.1
128	251 410 (5.36 %)	2:37:23	649.1	586.1
256	242 817 (5.18 %)	6:15:58	930.7	802.3

to discrete speeds and the BSP algorithm described in Section 5.1. To demonstrate this, we also consider instances with multi-edges to model speed adaptation—as was best practice in previous approaches [Bau+14, GP14, HF14]. We generate multi-edges in a rather conservative way, by sampling consumption functions at velocity steps of 10 km/h. Optimal paths (with respect to to the simple model) are then computed by the BSP algorithm. Indeed, we observe a significant speedup by simply switching to our more realistic model, as Table 5.5 indicates. It shows average figures for 100 queries with a rather small range (2 kWh). We see that TFP is up to two orders of magnitudes faster than BSP and finds paths that are up to 9.4 % quicker (within battery constraints), since it evaluates speed-consumption tradeoffs more fine-granularly while maintaining less query state (labels of continuous functions expressed by few parameters instead of large, discrete Pareto sets). This is interesting, as sampling was expressly considered to manage tractability [Bau+14, Bau+16e, GP14, HF14, SMS17]. In fact, even though atomic operations (linking and comparing labels) are more expensive for TFP, a drastic reduction in the number of vertex scans explains the speedup.

Evaluating Queries. In what follows, we focus on different variants of our fastest approach, CHAsp, since our basic algorithms are too slow for reasonable ranges (query times exceed our predefined threshold of one hour). For a comparison of CHAsp and basic algorithms, see Figure 5.23 discussed further below.

Table 5.6 shows details on CH preprocessing effort and its impact on query performance subject to different core sizes on Ger-PG, assuming a battery capacity of 16 kWh. Vertex contraction was stopped as soon as the average degree of active (i. e., contractable) vertices in the core reached a certain threshold. We report the resulting core

Table 5.7: Preprocessing and query performance (16 kWh). For each considered instance, we provide CH preprocessing times and query times of exact CHAsp, using the potential functions π_δ and π_φ, respectively. Reported query times are average values of 1 000 in-range queries, for a battery capacity of 16 kWh. We also show the number of settled labels (# Labels) and label comparisons during the forward search (# Dom.).

Inst.	Pr. [h:m:s]	Algo.	# Labels	# Dom.	Time [ms]
Ger-AX	30:33	CHAsp-π_δ	152	3 788	4.2
		CHAsp-π_φ	61	448	17.0
Ger-PG	31:43	CHAsp-π_δ	32 773	6 352 488	451.8
		CHAsp-π_φ	6 008	491 173	434.0
Eur-AX	3:10:43	CHAsp-π_δ	124	2 175	4.0
		CHAsp-π_φ	73	1 006	15.8
Eur-PG	3:09:22	CHAsp-π_δ	23 304	5 024 403	346.1
		CHAsp-π_φ	6 629	800 430	341.7

sizes and preprocessing times, as well as query times of our fastest exact algorithms. We see that contraction becomes much slower beyond a core degree of 32, which is explained by the small number of remaining active vertices. For instance, only 58 796 out of the reported 305 301 vertices in the core are active when the average degree reaches 32. This also explains why the speedup compared to the baseline (a threshold of 0 for the average degree of active core vertices yields plain TFP combined with A* search) is much smaller than in plain, single-criterion CH with scalar edge costs [Gei+12b]. Similar deteriorations in speedup were observed in other complex scenarios, such as time-dependent profile computation [Bat+13], time-dependent aircraft flight planning [Bla+16], and multicriteria routing [FS13]. Nevertheless, CH still yields an improvement by up to an order of magnitude in our case. In all experiments below, we pick an average core degree of 32 as stopping criterion of CH preprocessing. The resulting core size depends on different parameters, including vehicle range and error thresholds (of heuristic variants). Relative core sizes thus vary between 2.8 % for Ger-AX and 8.5 % for Eur-PG, which is explained by the difference in the amount of edges with negative consumption. Recall that this has a significant impact on the number of active vertices and the contraction order (see Section 5.3.5).

Table 5.7 shows the performance of CHAsp on all four considered instances for an EV with a battery capacity of 16 kWh. Note that we report vertex scans and dominance tests of the forward search only, excluding search spaces of the A* backward search (query times include both the forward and the backward search, though). In all cases, the optimal solution is found in well below a second on average. However, queries are significantly faster for the artificial model, where we achieve quite practical times in the order of milliseconds. This gap in running time is explained by the difference in the

Table 5.8: Preprocessing and query performance (85 kWh). We provide the same measures for the same instances and models as in Table 5.7, only this time running 1 000 random in-range queries with a higher battery capacity of 85 kWh.

Inst.	Pr. [h:m:s]	Algo.	# Labels	# Dom.	Time [ms]
Ger-AX	30:34	CHAsp-π_δ	24 715	4 312 923	552.3
		CHAsp-π_φ	406	11 813	1 236.7
Ger-PG	31:44	CHAsp-π_δ	2 272 350	2 130 447 427	131 562.0
		CHAsp-π_φ	32 182	6 836 380	14 873.5
Eur-AX	3:10:43	CHAsp-π_δ	27 358	12 159 343	960.9
		CHAsp-π_φ	871	46 529	1 174.7
Eur-PG	3:13:01	CHAsp-π_δ	—	—	—
		CHAsp-π_φ	105 792	44 986 403	34 617.4

number of edges with negative cost, induced by the underlying consumption models. One could even argue that the instances Ger-PG and Eur-PG are rather excessive in this regard, by not accounting for any auxiliary consumers at all. As a result, these instances are significantly more difficult to solve for our algorithms. Regarding the potential functions π_δ and π_φ, the search space is consistently smaller when using π_φ, but the backward search is more expensive. In fact, it becomes the major bottleneck for the considered battery capacity of 16 kWh on the easier instances. Consequently, query times are slowed down by about a factor of 4 in this case.

We also provide results for a larger battery capacity of 85 kWh, shown in Table 5.8. As before, the model with auxiliary consumers is much easier to solve. We obtain optimal results for long-range queries in less than a second on these instances. For the harder instances (no auxiliary consumers), the potential function π_φ provides better results due to its better scalability. Note that at least one query exceeded the maximum computation time of one hour on Eur-PG when using the potential function π_δ, hence no timings are reported. In summary, we can solve the problem examined in Section 5.3 *optimally* in less than a second on average for typical ranges, even on hard instances. For long ranges, our algorithm computes the optimal solution in well below a minute on the most difficult instance when using the potential function π_φ, despite its exponential worst-case running time.

In Table 5.9, we evaluate our heuristic approach on the difficult instances for different choices of the parameter ε (in % of total battery capacity; see Section 5.3.3). During preprocessing of CHAsp, new shortcuts are included only if they *significantly* improve on the existing ones. Thus, preprocessing becomes faster and core sizes (not reported in the table) decrease down to around 70 % of their original size. Query times also drop significantly: We achieve a considerable speedup by an order of magnitude. Regarding result quality, the choice of ε clearly matters. For $\varepsilon = 0.01$, the decrease in quality is

Table 5.9: Performance of the heuristic variant of CHAsp-π_δ, for different choices of the parameter ε (see Section 5.3.3) on Ger-PG and Eur-PG. We show figures on query performance for the same 1 000 random queries as in Table 5.7. Additionally, we report the percentage of feasible and optimal results, as well as the average and maximum relative error of all queries where a feasible solution was found.

			Query			Result Quality			
Ins.	Pr. [h:m:s]	ε	# Labels	# Dom.	T. [ms]	Feas.	Opt.	Avg.	Max.
Ger-PG	31:43	0.00	32 773	6 352 488	451.8	100.0 %	100.0 %	1.0000	1.0000
	30:41	0.01	19 922	1 949 458	225.6	100.0 %	89.4 %	1.0001	1.0047
	25:49	0.10	6 891	208 058	75.6	98.9 %	62.8 %	1.0013	1.0502
	17:48	1.00	1 742	11 149	30.7	95.1 %	47.6 %	1.0144	1.2294
Eur-PG	3:09:22	0.00	23 304	5 024 403	346.1	100.0 %	100.0 %	1.0000	1.0000
	3:04:48	0.01	12 803	1 132 685	151.6	100.0 %	82.8 %	1.0001	1.0145
	2:47:09	0.10	5 045	126 662	60.9	99.5 %	57.5 %	1.0020	1.0418
	2:14:03	1.00	1 428	7 641	28.2	92.7 %	45.8 %	1.0203	1.3960

negligible, but speedup (about a factor of 2) is moderate. For $\varepsilon = 0.1$, on the other hand, the optimal solution is still found in many cases. The average error is roughly 0.2 %, while the overall maximum is 5 %, which is acceptable in practice. Finally, for $\varepsilon = 1.0$, both the average and maximum error increase significantly. Given that speedup is limited compared to the case $\varepsilon = 0.1$, we conclude that the latter provides the best tradeoff in terms of quality and query performance. Providing high-quality solutions, it enables query times of well below 100 ms, which is fast enough even for interactive applications. Moreover, note that in cases where no path is found (about 1 % of all queries for $\varepsilon = 0.1$), a simple fallback solution could return the energy-optimal path, which can be computed quickly; see Chapter 4. For the easier instances Ger-AX and Eur-AX (not reported in the table), we generally observe smaller errors, but also less speedup. This can be explained by the fact that the A* backward search often becomes the bottleneck in the easier scenario when combined with a heuristic variant of TFP. In particular, using the more sophisticated potential function π_φ does not pay off in this case.

Evaluating Scalability. We evaluate our fastest exact algorithms following the methodology of Dijkstra ranks, defined for a query as the number of vertex scans when running Dijkstra's algorithm with costs representing unconstrained driving time [Bas+16, SS05]. Thus, higher ranks correspond to harder queries.

Figure 5.22 shows results for our fastest exact approaches on Eur-PG, assuming a battery capacity of 16 kWh. We ran 1 000 random queries per Dijkstra rank (note that these are not necessarily in-range queries). It turns out that median running

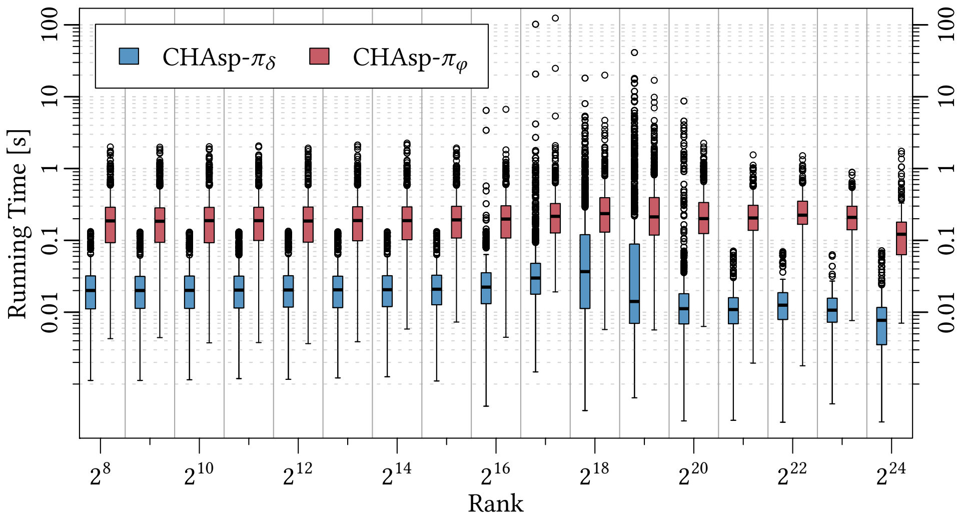


Figure 5.22: Running times of CHAsp subject to Dijkstra rank. We show results for both potential functions π_δ and π_φ. For each rank, we consider 1 000 random queries on Eur-PG, assuming a battery capacity of 16 kWh.

times of CHAsp-π_δ and CHAsp-π_φ are quite robust towards varying Dijkstra ranks. We obtain the most expensive queries at ranks 2^{17}–2^{19}. (Note that random in-range queries are likely to be among these most difficult ranks.) For higher ranks, the target is often unreachable. In most cases, this is detected by the backward searches for potential computation, as lower bounds on consumption exceed the battery capacity (see Section 5.3.4). Note that we could achieve further speedup for high ranks, by implementing reachability flags or running any technique that quickly computes energy-optimal routes to detect unreachable targets (see Chapter 4). For lower ranks (i. e., more local queries), the target is likely to be reachable on an unconstrained shortest path, so goal direction of the potential functions works very well. In such cases, the backward phase of A* search becomes the major bottleneck of the query, which explains why the lightweight potential π_δ yields better query times.

Although the median running time of CHAsp-π_φ is consistently higher than the median of CHAsp-π_δ, the former is also more robust in that it produces fewer outliers. Its more sophisticated potential function pays off especially for harder queries. Nevertheless, as worst-case running time is exponential, we observe a few outliers that exceed the median by several orders of magnitude.

While query times are relatively stable for different Dijkstra ranks, the vehicle range has a major influence on query performance. Therefore, we evaluate our approaches for different battery limits in Figure 5.23. For every considered battery capacity, we ran 100 random in-range queries. We report the median running time if each of the

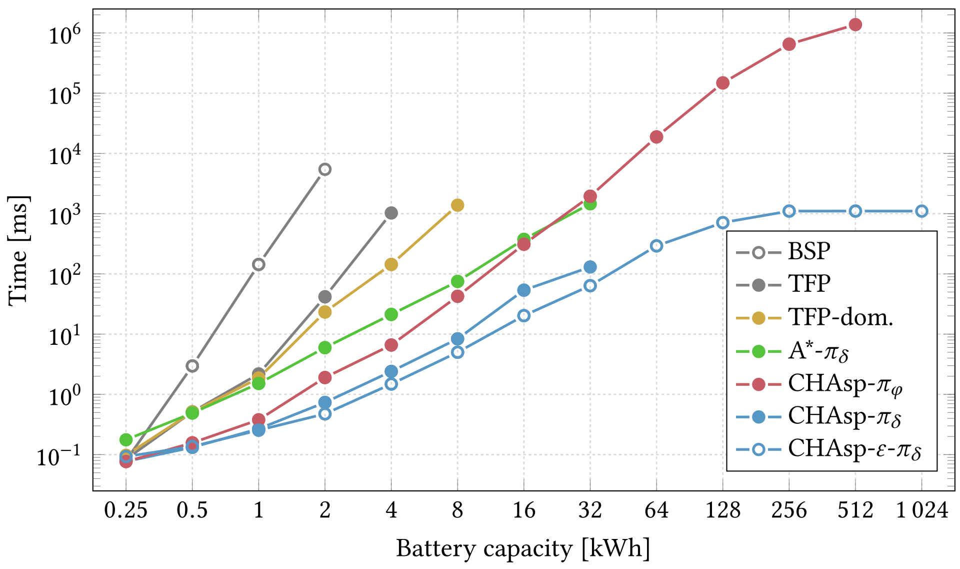


Figure 5.23: Running times for different battery capacities. For each considered capacity (ranging from 0.25 kWh to 1 024 kWh), the plot shows the median running time of 100 random in-range queries, provided that no query exceeded an hour of computation time. We evaluate the BSP algorithm using multi-edges corresponding to speed samples, the TFP algorithm using pairwise dominance tests, TFP with improved dominance tests (TFP-dom.), and our proposed speedup techniques (A^*-π_δ, CHAsp-π_δ, and CHAsp-π_φ). We also show our heuristic approach with parameter choice $\varepsilon = 0.1$, denoted CHAsp-ε-π_δ.

100 queries terminated within one hour. For small capacities, this enables our basic approaches, which are shown in the plot as well. We also evaluate the BSP algorithm, using multi-edges to model speed adaptation.

As before, we observe a considerable speedup by simply switching to our more realistic model, which is based on consumption functions. As discussed before, a vast reduction in the number of vertex scans more than makes up for the more expensive basic operations when using TFP. It achieves a speedup of up to two orders of magnitude over BSP. Plugging in the improved dominance checks described in Section 5.3.3 pays off as well, as it yields further speedup for battery capacities beyond 1 kWh. Adding A^* search, we achieve reasonable median running times of about a second for capacities of up to 32 kWh, without any preprocessing. Our technique CHAsp-π_δ adds preprocessing to provide further speedup by about an order of magnitude. Matching our previous observations, median running times of CHAsp-π_φ are slower for all ranges up to 32 kWh. However, this algorithm is more robust against outliers and is the only exact method that terminates within an hour for all queries at 64 kWh and up. As a result, we are able to compute *provably optimal* results for (hypothetical) ranges of up to 512 kWh (around 3 000 km) in less than an hour. Finally, our heuristic variant scales rather well with vehicle range. Query times actually bottom out for large battery capacities, as the vehicle range gets close to the graph diameter (for 1 024 kWh, the whole graph is always reachable).

5.5 Final Remarks

In this chapter, we proposed novel approaches for EV route planning that compute constrained shortest paths based on realistic models of energy consumption and charging stations. First, we introduced the CFP algorithm, which computes shortest feasible paths with charging stops, minimizing overall trip time. Second, our TFP algorithm respects battery constraints and takes adaptive speeds into account. Both approaches can be improved by nontrivial combination with vertex contraction and goal-directed search. Our resulting speedup techniques, CHArge and CHAsp, solve the underlying $\mathcal{NP}$-hard problems *optimally* and with practical performance, even on large, realistic inputs. For typical EV ranges, they find optimal solutions within seconds and below, making our techniques the first practical exact approaches—with running times similar to previous methods that are inexact or use less accurate models [Bau+14, GP14, HF14, Sto12a]. We also proposed heuristic variants that enable even faster queries, while offering high-quality solutions.

Future Work. An obvious next step would be the combination of both problem settings discussed in this chapter (and the algorithmic approaches to solve them) to finding routes with charging stops that allow for adaptive speeds [Bau+16e, Nik17]. Moreover, we are interested in more detailed models of energy consumption on turns. In preliminary experiments, we modified our input instances in accordance with a known edge-based approach for integrating turns [GV11]. Using a simple model that takes consumption overhead for acceleration and deceleration along turns (or when speed limits change) into account, we observed that preprocessing took slightly longer but also resulted in smaller cores, while query times remained nearly the same. This can be explained by the fact that the graph size increases when it is enriched with turn costs, but the number of nondominated solutions decreases at the same time (minor detours no longer allow energy savings). This indicates that our techniques can be readily extended to handle turn costs and turn restrictions.

For integration of historic and live traffic, the adaptation of customizable techniques appears to be a natural extension of our approaches. While preliminary experiments showed that our techniques based on CH perform equally fast on vertex orders required by CCH [DSW16], a major challenge is the design of fast customization algorithms that can handle the dynamic data structures required by label sets in our approach. On the other hand, preprocessing time for CHArge is already in the order of minutes and can even be reduced by increasing the core size at the cost of higher query times;

see Table 5.1 in Section 5.4.1. Similarly, preprocessing time of CHAsp can be reduced to about an hour on continental road networks; see Table 5.6 in Section 5.4.2. This is fast enough for many applications that require frequent updates of the graph.

We are also interested in extending our algorithms to more complex settings, e. g., by taking into account that charging stations might currently be in use by other customers [SDK17]. In such a scenario, some charging stations could also be reserved in advance to keep waiting times low [Mog15, QZ11]. Recall that CHArge keeps all charging stations in the core graph, so dynamic reservation systems could be integrated without any effect on the preprocessing routine.

From a practical point of view, it might be interesting to consider adaptive speeds only on the fastest roads (e. g., motorways), where going below the speed limit really pays off the most. Then, the majority of edge costs in the graph become constant, so contracting vertices incident to only constant edges in CH might be a promising approach. Given that vertices corresponding to motorways correlate with vertices of high CH rank in a natural way, this could yield quite practical running times.

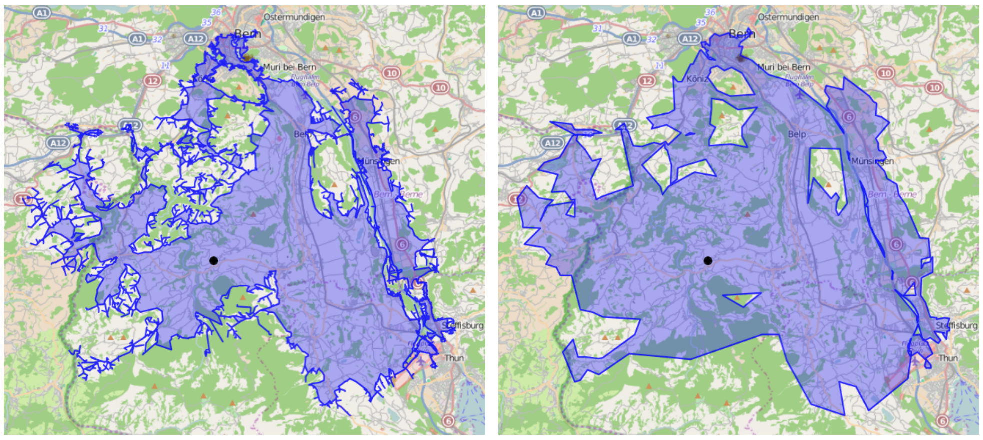
To enable faster heuristic variants, it would also be useful to precompute potentials for A* search, as in ALT [GH05, GW05]. From a more theoretical perspective, approximability of both problem settings considered in this chapter is an open question. Recent results by Strehler et al. [SMS17] include an FPTAS for a very similar problem concerning routes with both adaptive speeds and charging stops, which might extend to our setting. Regarding adaptive speeds, efficient representation and comparison of (general) bivariate SoC functions is an open issue. Similar to profile queries discussed in Chapter 4, one could also extend both problem settings considered in this chapter by asking for an optimal solution for *every* initial SoC with respect to a given source and a given target.

6 Fast Exact Visualization of Isocontours in Road Networks

How far can I drive my EV, given my position and the current SoC? This question expresses range anxiety (the fear of getting stranded) caused by limited battery capacities and a sparse charging infrastructure. An answer in the form of a map that visualizes the reachable region helps to find charging stations in range and to overcome range anxiety. This reachable region is bounded by curves that represent points of constant energy consumption; such curves are usually called *isocontours* (or *isolines*). Isocontours are typically considered in the context of functions $f\colon \mathbb{R}^2 \to \mathbb{R}$, e. g., if f describes the altitude in a landscape, then the terrain can be visualized by showing several isocontours (each representing points of constant altitude). In our setting, f would describe the energy necessary to reach a certain point in the Euclidean plane. However, f is actually defined only for a discrete set of points, namely for the vertices of the road network. Thus, we have to fill the gaps by deciding how the isocontour should pass through regions between the roads. The fact that the quality of the resulting visualization heavily depends on these decisions makes computing isocontours in road networks an interesting algorithmic problem.

Somewhat more formally, we assume the road network to be given as a directed graph $G = (V, E)$, along with vertex positions in the plane and two cost functions $d\colon E \to \mathbb{R}_{\geq 0}$ and $c\colon E \to \mathbb{R}$ representing length (or distance) and resource consumption, respectively. For a source vertex $s \in V$ and a range $r \in \mathbb{R}_{\geq 0}$, a vertex $v \in V$ belongs to the *reachable subgraph* if the shortest path from s to v has a total resource consumption of at most r. Note that shortest paths are computed according to the length, while reachability is determined by the consumption. Coming back to our initial question concerning the range of an EV, the source vertex is the initial position, the range is the current SoC, the length corresponds to driving time, and energy is the resource consumed on edges. We allow negative resource consumption to take recuperation into account. For the sake of simplicity, we consider the cost function d to be nonnegative in this chapter. Nevertheless, all algorithms described below can also be adapted to range visualization subject to driving on *energy-optimal* routes by the same means as described in Chapter 4.

Note that our setting is sufficiently general to also allow for other applications. By setting the length as well as the resource consumption of edges to the corresponding driving time (i. e., $d \equiv c$), one obtains the special case of *isochrones*. There is a wide range of applications for isochrones, including reachability analyses [Bau+08, Gam+11, GBI12, OMS00], geomarketing [Efe+13b], and environmental and social sciences [IBG13]. Known approaches focus on isochrones of small or medium range.



(a) (b)

Figure 6.1: Real-world example of isocontours in a mountainous area (near Bern, Switzerland). Both figures visualize the range of an EV positioned at the black disk with an SoC of 2 kWh. Note that the polygons representing the isocontour contain holes, due to unreachable high-ground areas. (a) An isocontour with over 10 000 segments computed by the approach described in Section 6.5.1, which resembles state-of-the-art techniques [MG10]. (b) The result of one of our new approaches, presented in Section 6.5.3, using 416 segments.

But isochrones can be useful in more challenging scenarios, for example, to visualize the area reachable by a truck driver within a day of work. Similarly, the range of an EV is beyond the scale of isocontours considered by previous approaches. This motivates our work on fast isocontour visualization.

We propose isocontours in road networks that are represented by *polygons*. Our algorithms for computing the isocontours are guided by three major objectives: (1) Isocontours must be *exact* in the sense that they correctly separate the reachable subgraph from the remaining unreachable subgraph; (2) the isocontours should be polygons of low *complexity* (i. e., consist of few segments, enabling efficient rendering and a clear, uncluttered visualization); and (3) algorithms should be fast enough in practice for interactive applications, even on realistic inputs of continental scale.

Figure 6.1 shows an example of isocontours visualizing the range of an EV. Figure 6.1a depicts a polygon that closely resembles the output of isocontour algorithms considered state-of-the-art in recent works [Efe+13a, Efe+13b, GBI12, MG10]. Unfortunately, the number of segments becomes quite large even in this medium-range example (more than 10 000 segments). In this chapter, we propose algorithms for efficiently computing polygons with fewer segments to represent isocontours in road networks. All approaches compute isocontours that are exact, i. e., they contain exactly the subgraph that is reachable within the given resource limit, while having low descriptive complexity. Figure 6.1b shows the result of our approach presented in

Section 6.5.3, which represents the same reachable subgraph as in Figure 6.1a, but uses only 416 segments in total.

Chapter Overview. In Section 6.1, we formalize the notion of reachable and unreachable subgraphs. Moreover, we state the precise problem and outline our algorithmic approach to solve it. It requires multiple steps, which are covered by the subsequent Sections 6.2–6.5.

First, Section 6.2 tackles the important subproblem of computing the reachable subgraph from a given source. We show how it can be solved by a variant of Dijkstra's algorithm. For better query performance, we propose several speedup techniques that enable fast computation of a compact representation of the reachable subgraph and are easy to parallelize. We describe a new algorithm based on CRP [Del+17] and a faster variant of isoGRASP [EP14]. Furthermore, we introduce novel approaches that combine graph partitions with variants of PHAST [Del+13b, DGW11].

Given the reachable and unreachable subgraph computed by any of the techniques listed above, Section 6.3 attacks the subproblem of computing *border regions*, i. e., polygons that represent the geometric boundaries of the reachable and unreachable subgraph. An isocontour must separate these boundaries.

In Section 6.4, we consider the special case of separating two hole-free polygons by a polygon with minimum number of segments. This problem can be solved in $O(n \log n)$ time [Wan91], where n is the total number of segments of both input polygons. We propose a simpler algorithm that uses at most two additional segments, runs in linear time, and requires a single run of a *minimum-link path algorithm.* We also propose a minimum-link path algorithm that is simpler than previous approaches [Sur86].

Section 6.5 extends these results to the general case, where border regions may have holes. Since the complexity of the resulting problem is unknown, we focus on efficient heuristics that work well in practice, but do not give guarantees on the complexity of the resulting isocontours.

Section 6.6 contains our extensive experimental evaluation on our main test instance and other large, realistic inputs. It demonstrates that all approaches are fast enough even for use in interactive applications, computing isocontours within a few milliseconds. We close with final remarks in Section 6.7.

6.1 Problem Statement and General Approach

Let $G = (V, E)$ be a graph, which we consider as a geometric network where vertices have a fixed position in the Euclidean plane and edges are represented by straight-line segments between their endpoints. As before, we assume that G is strongly connected. A source vertex $s \in V$ and a range $r \in \mathbb{R}_{\geq 0}$ together partition the network into two parts, one that is within range r from s, and the part that is not. An isocontour separates

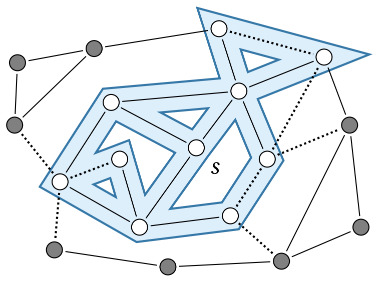


Figure 6.2: Graph with reachable (white) and unreachable (dark gray) vertices for a source s and range 2, assuming uniform edges costs of 1 (for length and consumption). Edges drawn solid are passable (unreachable) if both endpoints are reachable (unreachable). Dashed edges are border edges if they have an unreachable endpoint and accessible otherwise. Note that the blue polygon is in fact a range polygon.

these two parts. We are interested in visualizing such isocontours efficiently. Below, we give a precise definition of the (un)reachable parts of the network and formally define *range polygons*, which we use to represent isocontours (Section 6.1.1). Afterwards, we outline a generic approach to compute such an isocontour (Section 6.1.2).

6.1.1 Range Polygons

A path $P_{s,v}$ in G starting at the source $s \in V$ is *passable* if the consumption of $P_{s,v}$, i. e., the sum of its edge consumption values, is at most r. A vertex $v \in V$ is *reachable* (with respect to the range $r \in \mathbb{R}_{\geq 0}$) if the shortest s–v path is passable. A vertex that is not reachable is *unreachable*. For edges the situation is somewhat more complicated. We partition the edges into four types, namely unreachable edges, border edges, accessible edges, and passable edges. Figure 6.2 shows an example of the different edge types in a small graph. If both endpoints of an edge $(u,v) \in E$ are unreachable, then the edge (u,v) is also *unreachable*. If exactly one endpoint is reachable, then (u,v) is not part of the reachable network and we call it *border edge*. However, the fact that both u and v are reachable does not necessarily imply that (u,v) is part of the reachable network. Let $P_{s,u}$ and $P_{s,v}$ denote the shortest paths from s to u and v, respectively. If their resource consumptions do not allow traversal of the edge (u,v) in either direction, i. e., $c(P_{s,u}) + c(u,v) > r$ and $c(P_{s,v}) + c(v,u) > r$ in case $(v,u) \in E$, we do not consider (u,v) as reachable. Since we can reach both endpoints of (u,v), we call it *accessible*. Otherwise, the edge can be traversed in at least one direction, so it is *passable*.

Let V_r be the set of reachable vertices and let $V_u = V \setminus V_r$ be the set of unreachable vertices of G. Similarly, let E_u, E_b, E_a, and E_r denote the set of unreachable edges, border edges, accessible edges, and passable edges, respectively. Note that for arbitrary pairs of edges $(u,v) \in E$ and $(v,u) \in E$, both edges belong to the same set. The reachable part of the network is $G_r = (V_r, E_r)$, and the unreachable part is $G_u = (V_u, E_u)$. A *range polygon* is a plane (not necessarily simple) polygon P separating G_r and G_u in the sense that its interior contains G_r and has empty intersection with G_u. Note that every range polygon P intersects each border edge an odd number of times and each accessible edge an even number of times. In particular, an accessible edge may be totally or partially contained in the interior of a range polygon.

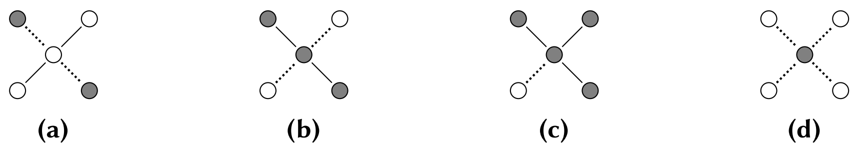


Figure 6.3: Different cases of crossing edges. (a) Intersection of a passable and an unreachable edge. The dummy vertex (center) is reachable. The unreachable edge is split into two border edges. (b) Intersection of an accessible and an unreachable edge. The dummy vertex is unreachable, dashed edges indicate new border edges. (c) Intersection of an unreachable edge and a border edge, creating unreachable edges and a new border edge. (d) An intersection of accessible edges creates an unreachable vertex and four new border edges.

If the input graph G is planar, one can construct a range polygon by first slightly *growing* the outer face of the subgraph induced by all reachable vertices. Then, *shrinking* the inner faces and making each shrunk face a hole results in a valid range polygon, though it may contain many holes; see the range polygon in Figure 6.2. However, if G is not planar, a range polygon may not even exist: If a passable edge crosses an unreachable edge, the requirements of including the passable and excluding the unreachable edge obviously contradict. To resolve this issue, we consider the planarization G_p of G, which is obtained from G by considering each intersection point $p \in \mathbb{R}^2$ as a *dummy vertex* that subdivides all edges of G containing p. We transfer the above partition of G into reachable and unreachable parts to G_p as follows. A dummy vertex is *reachable* if and only if it subdivides at least one passable edge of the original graph. As before, an edge of G_p is unreachable if both endpoints are unreachable, and it is a border edge if exactly one endpoint is reachable. If both endpoints are reachable, it is accessible (passable) if and only if the edge in G containing it is accessible (passable). Clearly, after the planarization, a range polygon always exists. Figure 6.3 shows different cases of crossing edges. Note that this way of handling crossings ensures that a range polygon for G_p contains the reachable vertices of G and excludes the unreachable vertices of G. However, unreachable edges of G may be partially contained in the range polygon if they cross passable edges.

Finally, to avoid explicit handling of special cases, we add a bounding box of dummy vertices and edges to G_p, connecting each vertex in the bounding box to its closest vertex in G_p with an edge of infinite length. Thereby, we ensure that neither the reachable nor the unreachable subgraph is empty, as the reachable (unreachable) subgraph contains at least the source (bounding box).

6.1.2 General Approach

We seek to compute a range polygon with respect to the planarized graph G_p that has the minimum number of holes, and among these we seek to minimize the complexity of the range polygon, i. e., its number of segments. Note that using G_p instead of G

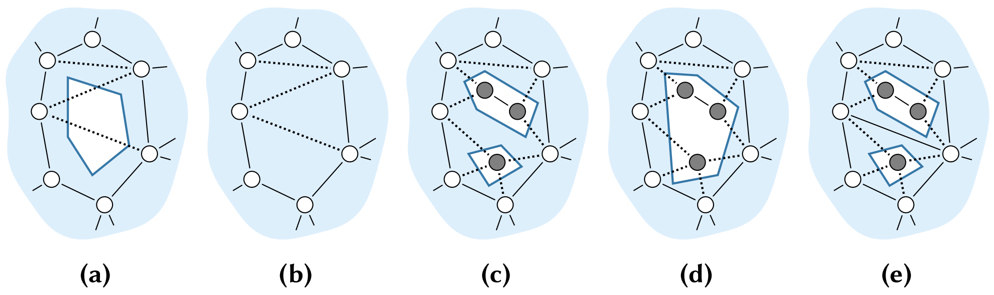


Figure 6.4: Removing unnecessary holes of a range polygon. (a) A hole that contains no unreachable vertices can always be removed. (b) The resulting interior (shaded area) of the range polygon. (c) Two holes that can be merged into one as they lie in the same border region. (d) The resulting range polygon. (e) These two holes cannot be merged, as they are separated by a passable edge.

may increase the number of holes (see the case depicted in Figure 6.3d), but guarantees the existence of a solution.

Consider the graph G' consisting of the union of the reachable graph G_r and the unreachable graph G_u. Clearly, all segments of the range polygon lie in faces of G'. A face of G' that is incident to both reachable and unreachable components is called *border region*. Since a range polygon separates the reachable and unreachable parts, each border region contains at least one connected component of a range polygon. Thus, the number of border regions is a lower bound on the number of holes. On the other hand, components in faces that are not border regions can be removed and multiple connected components in the same border region can always be merged, potentially at the cost of increasing complexity; see Figure 6.4. Hence, a range polygon with minimum number of holes (with respect to G_p) can be computed as follows.

1. Compute the reachable and unreachable parts of the input graph G.
2. Planarize the graph G, compute the reachable and unreachable parts of its planarization G_p.
3. Compute the border regions in G_p.
4. For each border region B, compute a simple polygon of minimum complexity that is contained in B and separates the unreachable components incident to B from the reachable component.

In the following sections, we discuss several alternative implementations for these steps. The first step is handled in Section 6.2. It is solved by a variant of Dijkstra's algorithm. We adapt speedup techniques to achieve faster queries in practice. Steps 2 and 3 are described together in Section 6.3. Section 6.4 and Section 6.5 are concerned

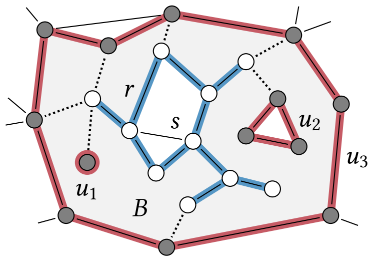


Figure 6.5: A border region B (shaded), defined by the set $R = \{r\}$ with the reachable boundary (blue) and a set $U = \{u_1, u_2, u_3\}$ of unreachable boundaries (red). The reachable boundary corresponds to the part that is reachable from the indicated source s with a range of 2, assuming uniform edge costs of 1 (with respect to both length and consumption).

with Step 4. Each connected component of the boundary of a border region is a hole-free, non-crossing polygon. Recall from Section 3.2 that such a polygon may contain the same segment twice in different directions or consist of a single vertex; see Figure 6.5. Each border region is defined by two sets R and U of hole-free, non-crossing polygons, where R contains the boundaries of the reachable components and U contains the boundaries of the unreachable components. We seek a simple polygon with the minimum number of segments that separates U from R. This problem has been previously studied. Guibas et al. [Gui+93] showed that it is $\mathcal{NP}$-hard in general. In our case, however, $|R| = 1$ always holds since the reachable part of the network is connected by definition. Guibas et al. left this case as an open problem and, to the best of our knowledge, it has not been resolved since.

In Section 6.4, we first consider border regions that are incident to only one unreachable component, i. e., $|R| = |U| = 1$. In this case, a polygon with the minimum number of segments that separates R and U can be found in $O(n \log n)$ time (where n is the total number of segments in the border region) using the algorithm of Wang [Wan91]. This algorithm is rather involved, but it contains a linear-time subroutine that computes an OPT + 1 approximation and is based on the computation of two minimum-link paths. Instead, we propose a simpler algorithm that uses at most two more segments than the optimum, runs in linear time, and relies on a *single* run of a minimum-link path algorithm. Clearly, spending an additional segment to save about a factor of 2 in running time is a favorable tradeoff in practice. Furthermore, we give a new linear-time minimum-link path algorithm that is simpler than previous algorithms for this problem [Sur86]. In Section 6.5, we consider the general case of our setting, where a border region may be incident to more than one unreachable component. We discuss several algorithms for this problem. As its complexity is unknown, we present heuristic approaches with (almost) linear running time that perform well in practice, but have no provable guarantees on the number of segments in the output.

6.2 Computing the Reachable Subgraph

We deal with the first important subproblem that we identified in our generic approach described in the previous section. It concerns the computation of the reachable and unreachable part of the input graph $G = (V, E)$. Interestingly, there is no canonical definition of a graph-based representation of isocontours in the literature. A unifying property, however, is the consideration of a range limit (time or some other limited resource), given only a source location for the query and no specific target. As a basic approach, we show that a pruned variant of Dijkstra's algorithm [Dij59] can be used to compute distances to all reachable vertices (Section 6.2.1). Newer approaches in the literature [EP14, Gam+11, GBI12] subscribe to the same model of computing the distances to reachable vertices. However, for our application it suffices to identify only the *set* of reachable vertices and edges, but no distances. In fact, it serves to just find the vertices and edges on the *boundary* of the range. Exploiting these observations, we derive new approaches for faster computation of isocontours.

We propose speedup techniques that employ offline preprocessing on the input graph G to quickly answer online queries consisting of a source vertex $s \in V$ and a range $r \in \mathbb{R}_{\geq 0}$. We distinguish *metric-independent* preprocessing (must be run when the input graph changes) and *metric-dependent* customization (only the length function and the consumption function change). We introduce techniques based on CRP (Section 6.2.2) and a faster implementation of isoGRASP (Section 6.2.3). Moreover, we introduce approaches that extend PHAST (Section 6.2.4). To simplify their description, we focus on the scenario of isochrones ($d \equiv c$) in Sections 6.2.1–6.2.4. Afterwards, we discuss how the algorithms can be modified to compute the necessary information in our more general scenario and serve output definitions from other applications that require, e. g., the set of reachable vertices (Section 6.2.5).

6.2.1 IsoDijkstra

As mentioned above, we consider the case $d \equiv c$ for now. In other words, we assume that length equals consumption on every edge, as is the case when computing, e. g., isochrones. Additionally, we focus on the computation of only the set E_b of border edges. Note that this set separates the reachable and the unreachable part, so it can be seen as a compact representation of these subgraphs. In Section 6.2.5, we describe how other information required for isocontour visualization is retrieved efficiently, such as the set of accessible edges. We also discuss then what has to be changed in the more general scenario, where distance and consumption of an edge may differ.

To distinguish, we call a border edge $(u, v) \in E_b$ *outward* if $c(s, u) \leq r$ and $c(s, v) > r$. Conversely, we call (u, v) *inward* if $c(s, u) > r$ and $c(s, v) \leq r$ hold. Our *isoDijkstra* algorithm works along the lines of Dijkstra's algorithm (see Section 3.3.1). It maintains and propagates *consumption labels* consisting of (tentative) values $c(\cdot)$ for resource consumption, all initially set to ∞, except for $c(s) = 0$ at the source $s \in V$. In each iteration, it extracts and settles a vertex $u \in V$ with minimum label $c(u)$ from a priority queue (initialized with s). It then scans all edges $(u, v) \in E$: If $c(u) + c(u, v) < c(v)$, it updates $c(v)$ accordingly and adds (or updates) v in the queue. Note that we can also adapt EVD in a similar fashion to deal with consumption values that represent energy consumption of an EV (see Section 4.2.1).

In our simplified problem setting (assuming $d \equiv c$), we can apply the following *stopping criterion*: The search may stop once the consumption label of the minimum element in the queue exceeds the range $r \in \mathbb{R}_{\geq 0}$. After its termination, we know the resource consumption of every settled vertex $v \in V$. Since consumption labels of unsettled vertices are upper bounds, v is reachable if and only if $c(v) \leq r$. Moreover, outward border edges are easily determined: We scan all vertices left in the queue, which must be unreachable, and add incident edges to E_b where the other endpoint is reachable. Inward border edges can be determined during the same scan over the queue if we apply the following modification to the isoDijkstra search. When settling a vertex $u \in V$, we also scan incoming edges $(v, u) \in E$. If $c(v) = \infty$, we insert v into the queue with a key of ∞. Thereby, we guarantee that for both types of border edges, the unreachable endpoint is contained in the queue when the search terminates.

6.2.2 IsoCRP

The three-phase workflow of CRP [Del+17] distinguishes preprocessing and metric customization. Building upon MLD [Del+09, HSW09, JP02, SWW00, SWZ02], the preprocessing phase finds a (multilevel) vertex partition of the road network with $L \in \mathbb{N}$ levels. For each level $\ell \in \{1, \dots, L\}$ of the partition, it induces an overlay graph H^ℓ containing all boundary vertices and boundary edges with respect to the partition $\mathcal{V}^\ell$ at level ℓ, as well as cliques of shortcut edges between pairs of boundary vertices that belong to the same cell $V_i^\ell \in \mathcal{V}^\ell$; see Section 3.3.2 for details. Metric customization computes the lengths of all shortcuts.

The basic idea of *isoCRP* is to run isoDijkstra on the overlay graphs during queries. We use shortcuts to skip cells that are entirely reachable, but *descend* into lower levels in cells that intersect the isocontour to determine border edges in G. There are two major challenges in determining such cells, which are illustrated in Figure 6.6. First, descending only into cells where traversing a shortcut exceeds the range $r \in \mathbb{R}_{\geq 0}$ is not sufficient: We may miss border edges that are part of no shortcut, but belong to shortest paths leading into the cell; see Figure 6.6a. We have to precompute additional information during customization to efficiently identify such cells in a query. Second, descents into cells must be consistent for all boundary vertices (i. e., we have to descend at all vertices). In particular, it is not sufficient to descend only at boundary vertices where a shortcut cannot be traversed; see Figure 6.6b. This motivates our two-phase approach, which ensures that we descend at either all or none of the boundary vertices of each cell. Below, we describe customization and the query phase of our approach.

Customization. Along the lines of plain MLD, we obtain shortcut lengths by running Dijkstra's algorithm from each boundary vertex, restricted to the respective cell.

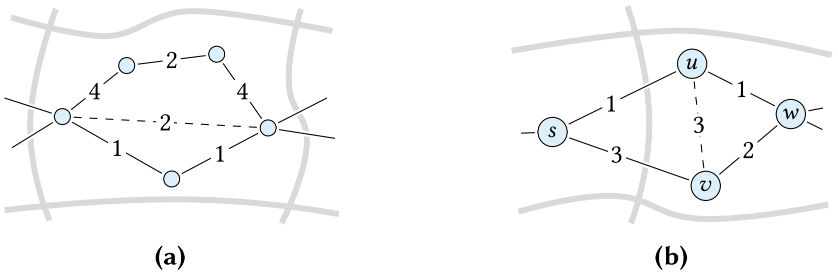


Figure 6.6: Determining active cells. Gray lines indicate cell boundaries of the partition. Dashed edges correspond to shortcuts. (a) The only shortcut of the cell has length 2. Thus, reaching either boundary vertex with a (remaining) range of $r \geq 2$ implies that the shortcut can be traversed, even if not all internal vertices are reachable. (b) Using the overlay, the search graph for a query from the source s consists of the (undirected) edges $\{s,u\}$, $\{s,v\}$, and $\{u,v\}$. An initial range of $r = 4$ is surpassed for the fist time after v is settled with consumption label 3, since the edge $\{u,v\}$ cannot be traversed from v without exceeding the range. Descending to the original graph only at v would lead isoDijkstra to erroneously report $\{v,w\}$ as border edge. The s–w path via u that makes w reachable and $\{v,w\}$ passable would not be found.

Additionally, we make use of the same searches to compute *eccentricities* for all boundary vertices. Eccentricities are used during queries to determine cells that may contain isocontour edges. Given a boundary vertex u in a cell V_i^ℓ, its (level-ℓ) eccentricity, denoted $\mathrm{ecc}_\ell(u)$ for $\ell \in \{1,\ldots,L\}$, is the maximum *finite* resource consumption necessary to reach some vertex $v \in V_i^\ell$ from u in the subgraph induced by V_i^ℓ. This subgraph is not strongly connected in general, i. e., there may exist vertices in V_i^ℓ that cannot be reached from u without leaving the cell V_i^ℓ. However, restricting eccentricities to cells is sufficient to maintain correctness of our approach and enables faster customization. Recall that the shortcuts of each cell form a clique, which is represented as a square matrix in contiguous memory for better cache efficiency. Storing eccentricities adds a single column to each matrix.

We compute eccentricities as follows. At the lowest level, the eccentricity of a boundary vertex $u \in V$ equals the consumption label of the last vertex that is scanned in the search from u, provided that no stopping criterion is applied (recall that the search is restricted to the level-1 cell V_i^1 containing u). On higher levels, previously computed overlays are used to obtain shortcuts for faster customization. We compute *upper bounds* on eccentricities for these levels. During the search from a boundary vertex $u \in V_i^\ell$ on the current level $\ell \in \{2,\ldots,L\}$, we maintain a label $\mathrm{ecc}_\ell(u)$. Initially, it is set to 0. When scanning a vertex $v \in V_i^\ell$, we check if the sum of the consumption label $c(v)$ and the eccentricity $\mathrm{ecc}_{\ell-1}(v)$ of v exceeds the current bound on $\mathrm{ecc}_\ell(u)$ and update it, if necessary. Observe that after the search has terminated, $\mathrm{ecc}_\ell(u)$ is indeed an upper bound on the level-ℓ eccentricity of u. It is not necessarily tight, however, since eccentricities of lower levels are restricted to their (lower-level) cells, whereas a shorter path from u to the corresponding endpoint may exist in the remaining graph. Upper bounds may lead to unnecessary descents into cells during queries, but they do not violate correctness and the overhead for computing the bounds during customization is negligible.

To improve data locality and simplify index mapping, vertices (which are represented by indices $\{1, \ldots, n\}$ in practice) are reordered in descending order of level during preprocessing, breaking ties by cell (see also Section 4.4.1).

Queries. Given a source vertex $s \in V$ and a range $r \in \mathbb{R}_{\geq 0}$, queries work in two phases. We say that a cell of the partition at some level is *active* if its induced subgraph contains at least one border edge. The first phase determines active cells, while the second phase descends into active cells to determine border edges. We allow the algorithm to falsely mark cells as active, which does not affect correctness (though the second phase may become more expensive).

The *upward phase* runs isoDijkstra on the search graph consisting of the union of the top-level overlay and, for each level of the multilevel partition, the subgraph of the overlay induced by the cell containing the source s. To determine active cells, we maintain two flags $\text{in}(\cdot)$ and $\text{out}(\cdot)$ per cell on each level, to indicate whether a cell contains at least one vertex that is reachable (*in range*) or unreachable (*out of range*), respectively. Initially, we set $\text{in}(V_i^\ell)$ to false and $\text{out}(V_i^\ell)$ to true for each cell V_i^ℓ of every level $\ell \in \{1, \ldots, L\}$. During the search, flags are updated as follows. When scanning some vertex $u \in V_i^\ell$ at level $\ell \in \{1, \ldots, L\}$ (i. e., $\ell \geq 1$ is the lowest level such that u is contained in the subgraph of H^ℓ that belongs to the search graph), we set $\text{in}(V_i^\ell)$ to true if $c(u) \leq r$. Next, we check whether $c(u)+\text{ecc}_\ell(u) \leq r$ holds. Observe that this condition is not sufficient to unset $\text{out}(V_i^\ell)$, because $\text{ecc}_\ell(u)$ was computed on the subgraph induced by V_i^ℓ. If this subgraph is not strongly connected, $c(u) + \text{ecc}_\ell(u)$ is not an upper bound on the resource consumption to any vertex in V_i^ℓ in general: A higher resource consumption may be necessary to reach vertices in V_i^ℓ that are not contained in the same strongly connected component as u in the cell-induced subgraph. However, we know that due to the matrix representation, vertices in different components are connected to u via a shortcut of length ∞. Therefore, when scanning an outgoing shortcut (u, v) with length ∞, we also check whether $c(v) + \text{ecc}_\ell(v) \leq r$. If the condition holds for u and all boundary vertices $v \in V_i^\ell$ that are not in the same strongly connected component as u in the cell-induced subgraph of V_i^ℓ, we can safely unset $\text{out}(V_i^\ell)$. Toggled flags are *final*, so we no longer need to perform any checks for them. After the upward phase finished, exactly the cells V_i^ℓ that have both $\text{in}(V_i^\ell)$ and $\text{out}(V_i^\ell)$ set to true are considered active, since border edges are only contained in cells that contain both reachable and unreachable vertices.

The *downward phase* consists of L subphases. In descending order of level, and for every active cell V_i^ℓ at the current level $\ell \in \{1, \ldots, L\}$, each subphase runs isoDijkstra on the subgraph of the overlay $H^{\ell-1}$ induced by V_i^ℓ (recall that $H^0 = G$ denotes the input graph). Initially, all boundary vertices of V_i^ℓ at level ℓ are inserted into the priority queue with their consumption labels according to the previous phase as key. For $\ell \geq 2$, we check eccentricities on-the-fly as before to mark active cells at level $\ell - 1$ for the next subphase. Each isoDijkstra search (including the upward phase) uses the stopping criterion from Section 6.2.1. Border edges are determined and added to the output at the end of each search, as described in Section 6.2.1. On higher levels, only boundary edges of the partition are added to E_b (in contrast to shortcuts, which do not belong to the input graph).

Parallelization. For faster customization, the cells of each level are processed in parallel [Del+17]. During queries, the (much more expensive) downward phase is parallelized in a natural way, as cells at the same level can be handled independently. We assign cells to threads and synchronize results between subphases (because each cell needs to be processed *after* its corresponding supercell). To reduce synchronization overhead, we process cells on lower levels in a top-down fashion within the same thread. This requires no synchronization constructs at all and false sharing (i. e., concurrent access to the same cache line) upon access of consumption labels becomes highly unlikely. Preliminary experiments showed that this strategy pays off for all but the topmost level, where the number of cells is small and hence, parallelism cannot be exploited well if all subcells of a top-level cell are assigned to the same thread.

Implementation Details. We make use of *clique flags* [Bau+16f] during queries, to reduce running time of the isoDijkstra searches on the overlays; c. f. Section 4.4.1. We store a flag for every boundary vertex $v \in V$ to indicate whether it was reached via a shortcut edge (as opposed to a boundary edge). When extracting v, we neither scan outgoing shortcut edges nor mark its cell as active if v was reached via a shortcut. This does not violate correctness, since scanning shortcut edges cannot improve any consumption labels in this case and the check for marking the cell was done before.

6.2.3 Faster IsoGRASP

The GRASP approach [EP14] extends MLD to batched query scenarios by storing for every level-ℓ boundary vertex, with $\ell \in \{0, \dots, L-1\}$, its incoming *downward shortcuts* from boundary vertices of its corresponding supercell at level $\ell+1$ (recall that at level 0, every vertex is a boundary vertex). Customization follows MLD, collecting downward shortcuts in a separate *downward graph* $H^\downarrow$. During queries, original isoGRASP [EP14] runs Dijkstra's algorithm on the overlays (as in MLD), marks all reachable top-level cells, and propagates distances in marked cells from boundary vertices to those at the levels below in a linear scan over the corresponding downward shortcuts. In this section, we accelerate isoGRASP significantly by making use of eccentricities.

Customization. Metric customization of our variant of isoGRASP works similar to isoCRP, computing shortcuts and eccentricities with Dijkstra's algorithm as described in Section 6.2.2. We obtain costs of downward shortcuts during the same searches. We apply edge reduction (removing shortcuts via boundary vertices) [EP14] to downward shortcuts, but stick to the matrix representation for overlay shortcuts.

Queries. As in isoCRP, queries run in two phases, with the *upward phase* being identical to the one described in Section 6.2.2. Afterwards, the *scanning phase* handles levels from top to bottom in L subphases to process active cells. Instead of running isoDijkstra on cell-induced overlays, each subphase performs a linear scan over the downward edges within each active cell on the current level. Given some active level-ℓ cell V_i^ℓ, with $\ell \in \{1, \dots, L\}$, we sweep over its *internal* vertices, i. e., all vertices of the overlay $H^{\ell-1}$ that lie in V_i^ℓ and are no level-ℓ boundary vertices. For each internal vertex $v \in V_i^\ell$, its incoming downward shortcuts are scanned to obtain the resource consumption that is necessary to reach v.

To determine active cells for the next subphase (in case $\ell \geq 2$), we maintain flags in$(\cdot)$ and out$(\cdot)$ as in isoCRP. When scanning some vertex $u \in V_i^{\ell-1}$, we set in$(V_i^{\ell-1})$ to true if $c(u) \leq r$ holds. To unset out$(V_i^{\ell-1})$, we check whether $c(u) + \mathrm{ecc}_{\ell-1}(u) \leq r$ and $c(v) + \mathrm{ecc}_{\ell-1}(v) \leq r$ hold for all boundary vertices $v \in V_i^{\ell-1}$ that are not in the same strongly connected component as u in the cell-induced subgraph of $V_i^{\ell-1}$ in the overlay $H^{\ell-1}$. We achieve some speedup by precomputing these vertices for each boundary vertex on each level and storing them in a separate adjacency array. After we updated the consumption labels of the internal vertices of V_i^ℓ, we also sweep over all level-ℓ boundary vertices once more to update the flags of their corresponding cells at level $\ell - 1$. Thereby, we make sure that we set flags correctly in cases where, e. g., the only reachable vertex in some cell at level $\ell - 1$ is a boundary vertex with respect to level ℓ (and not an internal vertex).

Similar to isoCRP, the upward phase reports all (original) border edges contained in its search graph. For the remaining border edges, we sweep over internal vertices a second time after processing a cell in the scanning phase. Note that at this point, reachability of these vertices is known. Hence, we scan incident (incoming and outgoing) edges of each internal vertex in the original graph G to determine missing border edges. To avoid duplicates and to ensure that endpoints of examined edges have correct consumption labels, we skip edges leading to vertices with higher indices. Since vertices are reordered during preprocessing, this automatically prevents us from checking edges leading to vertices on lower levels, where the consumption label might not be final yet. Moreover, edges on the same level are reported at most once, so we do not have to check for duplicates explicitly. In summary, an edge visited during this sweep is reported if it is present in the original graph, leads to a vertex with lower index, and reachability of both incident vertices differs.

Parallelization. Both customization and queries are parallelized in the same fashion as isoCRP. To avoid concurrent memory access requiring expensive locking mechanisms when customizing isoGRASP, we maintain thread-local containers for downward edges. The downward graph is built after customization is completed, by merging the edge sets of all threads.

6.2.4 IsoPHAST

Preprocessing of PHAST [Del+13b] contracts vertices in increasing order of (heuristic) importance, as in CH [Gei+12b]. Vertices are also assigned *levels* $\ell(\cdot)$, initially set to 0. When contracting a vertex $u \in V$, we set $\ell(v) = \max\{\ell(v), \ell(u) + 1\}$ for each uncontracted neighbor $v \in V$. Preprocessing results in two edge sets $E^\uparrow$ and $E^\downarrow$ consisting of *upward* and *downward* edges, each composed of original edges and shortcuts. PHAST handles one-to-all queries from a source $s \in V$ by running a *forward CH search* from s on $G^\uparrow = (V, E^\uparrow)$, followed by a *downward phase* consisting of a linear scan over all vertices in the graph in descending order of level. For each vertex, its incoming edges in $G^\downarrow$ are scanned to update consumption labels. RPHAST [DGW11] is tailored to one-to-many queries with target sets $T \subseteq V$. It first extracts the relevant subgraph $G_T^\downarrow$ of $G^\downarrow = (V, E^\downarrow)$ with respect to T. Then, it runs the linear scan on $G_T^\downarrow$; see Section 3.3.2 for details.

Our *isoPHAST* algorithm builds on PHAST and RPHAST to compute the reachable subgraph. Since the target set is not part of the input, we use graph partitions to restrict the subgraph that is examined for border edges. Queries work in three phases, in which we (1) run a forward CH search, (2) determine active cells, and (3) perform linear scans over all active cells as in PHAST. Below, we describe preprocessing of isoPHAST, before proposing different strategies to determine active cells.

Preprocessing. First, we find a (single-level) partition $\mathcal{V} = \{V_1, \ldots, V_k\}$ of the vertices of the input graph and reorder vertices, such that boundary vertices are pushed to the front, breaking ties by cell (providing the same benefits as in MLD). Afterwards, we use CH to contract all cell-induced subgraphs, but leave boundary vertices of the partition *uncontracted.* This results in a *core graph*, which is an overlay graph consisting of all boundary vertices, all boundary edges, and shortcuts between boundary vertices obtained during contraction. Non-core vertices inside cells (i. e., vertices that were contracted) are reordered according to their CH levels to enable cache-efficient linear downward scans. Our CH preprocessing routine follows Geisberger et al. [Gei+12b], but takes priority terms and hop limits from Delling et al. [Del+13b]. Preprocessing also results in an *upward graph* $G^\uparrow = (V, E^\uparrow)$, containing edges of each cell leading to vertices of higher level, added shortcuts between core vertices, and boundary edges of the partition $\mathcal{V}$. We also obtain a *downward graph* $G^\downarrow = (V, E^\downarrow)$ that stores for each non-core vertex its incoming edges from vertices of higher level. Further steps of preprocessing depend on the query strategy and are described below.

IsoPHAST-CD. Our first strategy, denoted *CD (Core-Dijkstra)*, performs isoDijkstra on the core graph to find active cells. As in isoCRP, we require eccentricities for core vertices to determine active cells. They are obtained during preprocessing as follows. To compute the eccentricity $\mathrm{ecc}(u)$ of a boundary vertex $u \in V_i$ in some cell $V_i \in \mathcal{V}$, we run (as last step of preprocessing) Dijkstra's algorithm on the subgraph of $G^{\uparrow}$ induced by all core vertices in the cell V_i of u, followed by a linear scan over the internal vertices of V_i to obtain their resource consumption, as in PHAST. When processing a vertex $v \in V_i$ during this scan, we update the eccentricity of u by setting it to $\mathrm{ecc}(u) = \max\{\mathrm{ecc}(u), c(v)\}$.

Queries start by running isoDijkstra from the source in $G^{\uparrow}$. Within the source cell, this corresponds to a forward CH search, since $G^{\uparrow}$ stores only upward edges. Note that for all other cells, only core vertices (i. e., boundary vertices of the partition) are visited. At core vertices, we maintain flags $\mathrm{in}(\cdot)$ and $\mathrm{out}(\cdot)$ to determine active cells, as described in Section 6.2.2. We use an adjacency array to store core neighbors in different cell-induced strongly connected components, as in Section 6.2.3. If the core is not reached, only the source cell is set active. Once the search has terminated, reachable core vertices have correct consumption labels. Next, we perform a linear PHAST scan over the internal vertices of each active cell, obtaining resource consumption from the source vertex to all vertices in active cells that are reachable (due to the stopping criterion used in the first phase, we only get upper bounds for unreachable vertices, which suffices for border edge detection).

Border edges crossing cell boundaries are added to the output set E_b during the isoDijkstra search, whereas border edges incident to at least one non-core vertex are obtained during the linear scans as follows. When scanning incident edges of a vertex $v \in V$, neighbors at higher levels have final consumption labels. Moreover, the label of v is final after scanning its incoming edges $(u, v) \in E^{\downarrow}$. Thus, looping through the incoming original edges a second time suffices to find the remaining border edges. Since original outgoing edges $(v, w) \in E$ to vertices $w \in V$ at higher levels are not contained in $E^{\downarrow}$ in general, we add dummy edges of length ∞ to $E^{\downarrow}$ to ensure that neighbors in G are also adjacent in $G^{\downarrow}$. Finally, we indicate with an additional flag whether an edge is a shortcut or an original edge to ensure that only original edges are added to E_b.

IsoPHAST-CP. Instead of running isoDijkstra, our second strategy *CP (Core-PHAST)* performs a linear scan over the core to find active cells. Eccentricities are precomputed after the generic preprocessing routine as described for isoPHAST-CD. Next, we use CH preprocessing to contract all remaining vertices in the core and reorder core vertices according to their levels (vertices with higher levels are pushed to the front). Finally, we add the shortcuts computed during core contraction to $G^{\uparrow}$ and $G^{\downarrow}$.

Queries strictly follow the three-phase pattern discussed above. We first run a forward CH search in $G^{\uparrow}$ (until the queue runs empty). Then, we determine active cells and compute resource consumption for all core vertices in a linear PHAST scan over the core, scanning edges in $G^{\downarrow}$ to propagate consumption values to vertices with lower level. As before, we maintain flags $\mathrm{in}(\cdot)$ and $\mathrm{out}(\cdot)$ for core vertices (c. f. Section 6.2.2) and use an adjacency array storing core neighbors in different cell-induced strongly connected components (c. f. Section 6.2.3). To find border edges between core vertices, we insert dummy edges into the core to preserve adjacency with respect to the original graph G. The third phase (linear scans over active cells) is identical to isoPHAST-CD.

IsoPHAST-DT. Both strategies described before require a relatively expensive second phase (determining active cells). Our third strategy *DT (Distance Table)* uses a distance (bounds) table to accelerate this phase. Working with such tables instead of a dedicated core search benefits from *edge partitions*, since the unique assignment of edges to cells simplifies border edge retrieval (recall that previous strategies had to collect border edges during both the second and the third phase of the algorithm). Given a partition $\mathcal{E} = \{E_1, \dots, E_k\}$ of the set E of edges in the input graph, the table stores for each pair E_i, E_j of cells, with $i \in \{1, \dots, k\}$ and $j \in \{1, \dots, k\}$, a lower bound $\underline{c}(E_i, E_j)$ and an upper bound $\bar{c}(E_i, E_j)$ on the resource consumption from E_i to E_j, i. e., it holds that $\underline{c}(E_i, E_j) \leq \mathrm{dist}_c(u, v) \leq \bar{c}(E_i, E_j)$ for all vertices $u \in E_i$ and $v \in E_j$ (we abuse notation, saying that $u \in E_i$ if the vertex u is an endpoint of at least one edge $e \in E_i$). During a query, given a source $s \in E_i$ for some $i \in \{1, \dots, k\}$ (if s is ambiguous with respect to $\mathcal{E}$, we pick any cell containing s) and a range $r \in \mathbb{R}_{\geq 0}$, all cells E_j with $j \in \{1, \dots, k\}$ and $\underline{c}(E_i, E_j) \leq r < \bar{c}(E_i, E_j)$ are set active.

Preprocessing first follows isoPHAST-CP, with three differences: (1) We use an edge partition instead of a vertex partition; (2) *unrestricted* eccentricities are computed on the *backward* graph of the core, with searches that are not restricted to cells but stop when all boundary vertices of the current cell are reached (these eccentricities are used below to obtain distance bounds during queries); (3) after computing eccentricities, we recontract the whole graph using a regular CH order (i. e., contraction of core vertices is not delayed), leading to sparser graphs $G^{\uparrow}$ and $G^{\downarrow}$. Afterwards, to quickly compute (not necessarily tight) consumption bounds for the tables, we run for each cell $E_i \in \mathcal{E}$ a (multi-source) forward CH search in $G^{\uparrow}$ from all boundary vertices of E_i. Then, we perform a linear PHAST scan over $G^{\downarrow}$, keeping track of the minimum and maximum consumption label per target cell. This yields, for all $i \in \{1, \dots, k\}$ and $j \in \{1, \dots, k\}$, lower bounds $\underline{c}(E_i, E_j)$ on the resource consumption from E_i to E_j, as well as upper bounds on the resource consumption from *boundary* vertices of E_i to E_j. To obtain the desired bound $\bar{c}(E_i, E_j)$, we increase the latter value by the maximum consumption when going from any vertex in E_i to a boundary vertex of E_i. This maximum consumption of E_i equals the maximum unrestricted eccentricity of all boundary vertices of E_i on the backward graph (which we computed before). As last step of preprocessing, we extract and store the relevant search graph $G_i^{\downarrow}$ for each cell $E_i \in \mathcal{E}$. This requires a target selection phase as in RPHAST for each cell E_i, using all (i. e., both distinct and ambiguous) vertices of E_i as input.

Queries start with a forward CH search in $G^{\uparrow}$. Active cells are determined independently in a scan over one row of the distance table (corresponding to the source cell). Let $E_i \in \mathcal{E}$ be the cell containing the source vertex $s \in V$ (if s is ambiguous, we can pick an arbitrary cell containing s). We mark a cell $E_j \in \mathcal{E}$ as active if and only if $\underline{c}(E_i,E_j) \leq r < \bar{c}(E_i,E_j)$. The third phase performs a linear RPHAST scan over $G_i^{\downarrow}$ for each active cell $E_i \in \mathcal{E}$, obtaining resource consumption values for all its vertices. Note that, although vertices can be contained in multiple search graphs, consumption labels do not need to be reinitialized between different scans, since the source remains unchanged. To output border edges, we proceed as before, looping through incoming downward edges twice (again, we add dummy edges to $G_i^{\downarrow}$ for correctness). To avoid duplicates (due to vertices that are contained in multiple search graphs), edges in $G_i^{\downarrow}$ have an additional flag to indicate whether the edge belongs to E_i. Border edges are reported only if this flag is set and they are contained in the original graph (which is indicated by another flag).

As mentioned above, search graphs $G_i^{\downarrow}$, with $i \in \{1,\ldots,k\}$, may share vertices, which increases memory consumption and slows down queries. For example, the vertex with maximum level is contained in *every* search graph. For better performance in practice, we use *search graph compression*, where we store the topmost vertices of the hierarchy (and their incoming edges) in a separate graph $G_c^{\downarrow} = (V_c, E_c^{\downarrow})$ and remove them from all graphs $G_i^{\downarrow}$. During queries, we first perform a linear scan over $G_c^{\downarrow}$ (obtaining resource consumption for all vertices $v \in V_c$), before processing search graphs of active cells. The size of $G_c^{\downarrow}$ is a tuning parameter.

Parallelization. Independent of the strategy, the first preprocessing steps are executed in parallel in a natural way by assigning cells of the partition to different threads, namely, extracting the cell-induced subgraphs for efficient (thread-local) contraction, contracting the non-core vertices, inserting dummy edges, and reordering non-core vertices by level. Afterwards, threads are synchronized and the search graphs $G^{\uparrow}$ and $G^{\downarrow}$ are built sequentially. Eccentricities are again computed in parallel (one thread per cell). For isoPHAST-CD and isoPHAST-CP, threads operate on distinct vertex sets. Consequently, there are no concurrent accesses to consumption labels and we can share labels among threads. However, each thread needs its own priority queue. For isoPHAST-DT, searches for unrestricted eccentricities are not limited to their cells. Therefore, every thread operates on its own set of consumption labels to avoid concurrent accesses. Our CH preprocessing is sequential, thus the core graph is contracted in a single thread (if needed). Computation of distance bounds is parallelized (one thread per cell, if needed).

Considering queries, the first two phases are run sequentially. Both isoDijkstra and the forward CH search are rather difficult to parallelize. Executing PHAST on the core in parallel does not pay off (the core is rather dense, resulting in many levels and thus, limited potential for parallelization). Distance table operations, on the other hand, are very fast and parallelization is not necessary. In the third phase, however, active cells can be assigned to different threads. Again, we share distance labels among threads, except for isoPHAST-DT, where search spaces may overlap and each thread uses its own distance labels to prevent concurrent accesses. Moreover, each thread runs its own forward CH search to initialize its labels in isoPHAST-DT.

Running the third phase in parallel can make the second phase of isoPHAST-CP a bottleneck. Therefore, we alter the way of computing flags $\mathrm{in}(\cdot)$ and $\mathrm{out}(\cdot)$ for this variant as follows. Initially, both flags of every vertex are set to false. After scanning a vertex $v \in V_i$ in some cell $V_i \in \mathcal{V}$, we now set $\mathrm{in}(V_i)$ if $c(v) \leq r$ and $\mathrm{out}(V_i)$ if $c(v) + \mathrm{ecc}(v) > r$. These checks are less accurate (more flags are toggled), but we no longer have to check boundary vertices in different strongly connected components induced by cells. Correctness of isoPHAST-CP is maintained, as no stopping criterion is applied and $\max_{v \in V_i} c(v) + \mathrm{ecc}(v)$ is a valid upper bound on the resource consumption from the source to each vertex in V_i. Hence, no active cells are missed.

To further accelerate the linear scans of isoPHAST-CD and isoPHAST-CP, we take the computer architecture of modern machines into consideration. Nowadays, most multi-socket systems have more than one NUMA (Non-Uniform Memory Access) node. Processors in such systems can access memory assigned to their NUMA node faster than memory assigned to different NUMA nodes. To exploit such computer architectures, we store the downward graph $G^{\downarrow}$ once on each NUMA node. During queries, each core uses the copy on the NUMA node its processor is assigned to.

6.2.5 Alternative Outputs

In this section, we describe modifications to deal with the general case of our problem setting, where length and resource consumption of edges may differ. Moreover, we presume in all subsequent sections of this chapter that we know

- for each vertex $v \in V$, whether v is reachable from the source $s \in V$;
- the set $E_x := E_b \cup E_a$ of all border edges and accessible edges;
- for every edge $(u, v) \in E \setminus E_x$, whether (u, v) is passable or unreachable.

Therefore, we describe a variant of isoDijkstra that computes the set E_x instead of E_b and handles the more general case mentioned above. Afterwards, we discuss necessary changes to our speedup techniques to cover the general case and the modified output. For the sake of simplicity, we assume that shortest paths are unique (with respect to the length function d). Thereby, we avoid the special case of two paths with equal distance but different consumption, which requires additional tie breaking and slight modifications to our data structures when computing the range of an EV [Buc15].

Generalizing IsoDijkstra. For a given source vertex $s \in V$ and a range $r \in \mathbb{R}_{\geq 0}$, the general variant of isoDijkstra maintains vertex labels consisting of (tentative) values $d(\cdot)$ for distance and $c(\cdot)$ for resource consumption, both initially set to 0 for s and ∞ for all other vertices. The algorithm uses a priority queue of vertices, initially containing s. In each step, it extracts the vertex $u \in V$ with minimum distance from the queue. Then, all outgoing edges $(u,v) \in E$ are scanned, checking for each whether $d(u) + d(u,v) < d(v)$ holds. If this is the case, the labels at v are updated accordingly to $d(v) = d(u) + d(u,v)$ and $c(v) = c(u) + c(u,v)$. Also, v is inserted into the queue if it is not contained already. Correctness follows directly from the fact that we simply sum up consumption values along shortest paths.

The set E_x of border edges and accessible edges can be computed on-the-fly as follows. Consider an arbitrary edge $(u,v) \in E$. We know whether (u,v) belongs to E_x as soon as both u and v were settled (and thus have final labels). Therefore, after extracting a vertex $u \in V$ from the queue, we check all incoming and outgoing edges and add them to E_x if the respective neighbor was settled and one of the following conditions holds: Either, exactly one endpoint of the edge is currently reachable, or both endpoints are reachable but the edge is not passable, i. e., $c(u) + c(u,v) > r$ if $(u,v) \in E$ and $c(v) + c(v,u) > r$ if $(v,u) \in E$. After termination, an edge $(u,v) \in E \setminus E_x$ is passable if u (and thus, also v) is reachable, and it is unreachable otherwise. Moreover, we know that a vertex $v \in V$ is reachable if and only if $c(v) \leq r$ holds.

Stopping Criterion. Note that the stopping criterion described in Section 6.2.1 no longer applies, since vertices are scanned in increasing order of distance rather than resource consumption. In general, simply pruning the search at unreachable vertices does not preserve correctness either. To see this, assume that we prune the search by not scanning any outgoing edges of an unreachable vertex $u \in V$ at some point during the search. Consider another unreachable vertex $v \in V$, such that the shortest s–v path contains u (and in particular, one of its outgoing edges). There might exist some (non-shortest) path from s to v with lower resource consumption that is found by the algorithm instead. Then, the vertex v is falsely identified as reachable.

Nevertheless, we can safely abort the search as soon as no reachable vertex is left in the priority queue, since no reachable vertex can be found from an unreachable vertex. (In case of negative consumption values, we presume that battery constraints apply; see Section 4.1.1.) To efficiently check whether this stopping criterion is fulfilled, we simply maintain a counter to keep track of the number of reachable vertices in the queue. As soon as it reaches 0, we abort the search. As in the simpler case discussed in Section 6.2.1, we scan vertices left in the queue after the search has terminated to determine any remaining border edges.

Adapting Speedup Techniques. We make the following modifications to maintain correctness of our speedup techniques in the general scenario. First of all, we slightly alter the definition of the eccentricity $\mathrm{ecc}_\ell(v)$ of a vertex $v \in V$ at level $\ell \in \{1, \ldots, L\}$, to the maximum finite resource consumption on any *shortest* path (with respect to the distance function d) from v to to any vertex inside the subgraph induced by the cell containing v at level ℓ.

As before, we maintain two flags $\mathrm{in}(V_i^\ell)$ and $\mathrm{out}(V_i^\ell)$ for each cell V_i^ℓ at every level $\ell \in \{1, \ldots, L\}$. The flag $\mathrm{in}(V_i^\ell)$ is set permanently as soon as a vertex $v \in V_i^\ell$ is settled and $c(v) \le r$ holds. However, updating the flag $\mathrm{out}(V_i^\ell)$ is more involved in the general scenario: We may unset it if the condition $c(v) + \mathrm{ecc}_\ell(v) \le r$ holds for v and all vertices in other cell-induced strongly connected components, but since vertices are no longer scanned in increasing order of resource consumption by isoDijkstra, the flag is not final until *all* boundary vertices of the cell were settled. For techniques based on isoDijkstra searches, we check the condition during each vertex scan as before and update the flag accordingly, i. e., the flag $\mathrm{out}(V_i^\ell)$ is set to false if $c(v) + \mathrm{ecc}_\ell(v) \le r$ holds for the settled vertex $v \in V_i^\ell$ as well as all vertices in the remaining cell-induced components, otherwise it is set to true. Consequently, the flag may be toggled multiple times throughout the search. To maintain correctness, we have to ensure that all boundary vertices that are (temporarily) considered as reachable (possibly via non-shortest paths) are settled eventually; see Figure 6.7. To this end, we relax the stopping criterion of isoDijkstra as follows. We maintain an upper bound $\bar{d} \in \mathbb{R}_{\ge 0}$ on the distance from s to any vertex that is currently reachable, initially set to $\bar{d} = 0$. Whenever we update the consumption label of a vertex $v \in V$ to some value $c(v) \le r$ after scanning an edge with head vertex v, we update the bound to $\bar{d} = \max\{\bar{d}, d(v)\}$. We stop the search as soon as (1) no reachable vertex is left in the queue and (2) the minimum key in the priority queue exceeds $\bar{d}$.

Outputs. After running the query algorithm of any proposed speedup technique, we can check in constant time (more precisely, linear time in the constant L) if a vertex is reachable as follows. Consider the (top-level) cell V_i^L of a vertex $v \in V$. If $\mathrm{in}(V_i^L)$ is not set, the cell contains no reachable vertices and v must be unreachable. Similarly, if $\mathrm{out}(V_i^L)$ is not set, the cell contains no unreachable vertices, and thus v is reachable. If both flags are set, we recursively proceed with the same check for the cell containing v on the level below. If both flags are set for the cell on level 1 containing v, we check if the consumption label of v exceeds the range. In the same manner, we determine

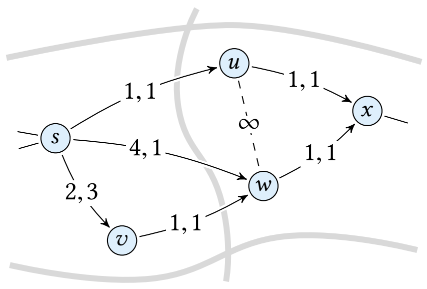


Figure 6.7: Finding active cells in the general scenario. Edge labels show their length and consumption (in this order). In an isoCRP query from s with range $r = 2$, all vertices in the cell of u are considered reachable when it is settled. Next, v is settled and w is updated as unreachable. At this point, no reachable vertices are left in the queue, so isoCRP misses the border edge (w, x) unless we adapt the stopping criterion.

whether an edge $e \in E \setminus E_x$ is passable or unreachable by inspecting reachability of its endpoints. Finally, to output the set E_x instead of E_b, we modify the definition of consumption-based eccentricities slightly, to the maximum finite sum of resource consumption of any vertex on a shortest path (from a given boundary vertex) and the resource consumption of an outgoing edge of this vertex. Thereby, we ensure that we also descend into cells that contain accessible edges but no border edge. Then, the set E_x is retrieved in a similar way as the set E_b.

Note that we represent the set of reachable vertices and all edge sets except E_x implicitly. This suits the problem considered in this chapter, but may be insufficient in other applications. However, all speedup techniques presented in this section can easily be adapted to produce a variety of other outputs, without increasing their running times significantly. As an example, we can modify our algorithms to output a list of all reachable vertices. A straightforward approach performs a linear scan over all vertices and determines the reachable ones as described above. We can do better by collecting reachable vertices on-the-fly: During isoDijkstra searches and when scanning active cells, we output each settled vertex that is reachable. In the scanning phase, we also add all internal vertices of cells V_i^ℓ for which $\mathrm{out}(V_i^\ell)$ is not set.

6.3 Computing the Border Regions

Given the reachable and unreachable part of the input graph, we describe the planarization of these parts and the extraction of all border regions. First, we map the information about the reachable and unreachable part computed by isoDijkstra (or a speedup technique) to the planarized graph G_p in $\mathcal{O}(m)$ time (Section 6.3.1). We then extract border regions by traversing all faces of the planar graph that contain border edges (Section 6.3.2). This requires linear time in the size of all border regions.

6.3.1 Planarization of the Input Graph

We planarize $G = (V, E)$ during the preprocessing stage to obtain $G_p = (V_p, E_p)$. Since edge costs and directions are irrelevant for all subsequent steps of our algorithms, we

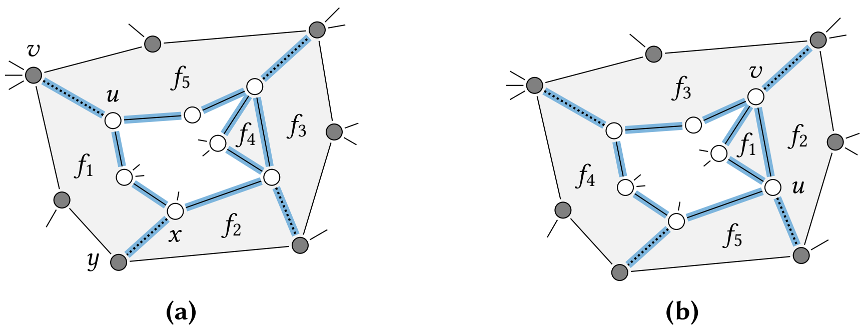


Figure 6.8: Visited edges (blue) when extracting a reachable boundary. (a) Starting at the border edge $\{u,v\}$, the face f_1 is traversed first until the border edge $\{x,y\}$ is encountered. Afterwards, the faces f_2, f_3, f_4, f_3, f_5 are visited in this order until the edge $\{u,v\}$ is reached again. (b) Starting at the accessible edge $\{u,v\}$, the face f_1 is traversed until the edge $\{u,v\}$ is reached again (on the same side). Then, the faces f_2, f_3, f_4, f_5, f_2 are processed before the other side of the edge $\{u,v\}$ is reached.

consider G_p to be undirected and do not store any cost functions with it to reduce space consumption. Our implementation uses the well-known sweep line algorithm [Ber+08, BO79] to compute G_p. It runs in $O((n + k) \log n)$ time, where k is the number of edge crossings. More involved algorithms with better asymptotic running times exist [Bal95, EGS10]. However, the value of k is usually small for road networks and somewhat higher (metric-independent) preprocessing effort is not an issue in our scenario. With each vertex in G_p, we store its original vertex in G (if it exists). In practice, where vertices are represented by indices $\{1, \dots, n\}$, this mapping can be done implicitly, since V_p is a superset of V.

During a query, after computing the reachable subgraph of the original graph G as described in Section 6.2, we compute reachability of dummy vertices and the set E_x of border edges and accessible edges in G_p as follows. First, we have to ensure that border edges and accessible edges returned by isoDijkstra are actually contained in G_p. We add a flag to each edge in G during preprocessing that indicates whether an edge $e \in E$ is also contained in G_p. We modify isoDijkstra (and the speedup techniques) described in Section 6.2, such that edges are added to E_x only if this flag is set.

After isoDijkstra terminates, we check for each dummy vertex $v \in V_p$ whether it is reachable, by checking passability of all original edges in E that contain v. To this end, we precompute an array of all original edges that were split during planarization, and for each split edge a list of dummy vertices it contains (an original edge may intersect multiple other edges). In a query, we scan the array of edges and mark for each passable edge its dummy vertices as reachable. Finally, we perform a linear scan over all edges in G_p that have at least one dummy vertex as endpoint to determine any missing edges in E_x. To test whether some edge in G_p is accessible, we store pointers to the original edges containing it. Note that these scans produce limited overhead in practice, since the number of dummy vertices in graphs representing road networks is typically small (large parts of the input are planar to begin with). Afterwards, a vertex in V_p is reachable if it is a dummy vertex marked as reachable, or its corresponding original vertex is reachable. Otherwise, it is unreachable. For edges contained in $E_p \setminus E_x$, we check reachability of their endpoints to determine whether they are passable or unreachable.

An alternative approach modifies isoDijkstra to work directly on a (directed) planar graph G_p to avoid additional linear scans. However, this produces overhead during the search (e. g., case distinctions for dummy vertices). Consequently, such approaches did not provide significant speedup in preliminary experiments. Moreover, determining the reachable subgraph of G_p in a separate step simplifies the integration with speedup techniques discussed in the previous section.

6.3.2 Extracting Border Regions

Given the set E_x of border edges and accessible edges in G_p, we describe how the actual border regions are computed (i. e., the polygons describing R and U). We traverse all faces of G_p that contain edges in E_x and collect the segments that form boundaries of the border regions. Clearly, all passable edges in these faces are part of some reachable boundary, while all unreachable edges belong to an unreachable boundary. Moreover, since G_p is connected, all faces contained in a border region must contain an edge in E_x. Thus, traversing these faces is sufficient to obtain all border regions.

In somewhat more detail, we maintain two flags for every edge $\{u, v\}$ in E_x indicating whether u or v has been visited, respectively, each initially set to false. Let $\{u, v\}$ be the first edge of E_x that is considered, and without loss of generality, let u be reachable. We compute the (unique) reachable component $R_{\{u,v\}}$ of the border region $B_{\{u,v\}}$ containing $\{u, v\}$; see Figure 6.8. We mark u as visited and traverse the face to the left of $\{u, v\}$, following the unique neighbor w of u in this face that is not v. Every edge that we traverse is added to $R_{\{u,v\}}$. As soon as we encounter an edge $\{x, y\} \in E_x$, we continue by traversing the *twin face* of $\{x, y\}$, i. e., the unique face of G_p that contains the other side of $\{x, y\}$. The edge $\{x, y\}$ itself is not added to $R_{\{u,v\}}$, but we mark the reachable endpoint x that was added to $R_{\{u,v\}}$ in the previous step as visited. The current extraction step is finished as soon as the other side of $\{u, v\}$ is reached; see Figure 6.8. If v is unreachable, $\{u, v\}$ is a border edge. Thus, we continue with the extraction of the unreachable component $U_{\{u,v\}}$ containing v in the same manner and assign it to $B_{\{u,v\}}$.

We loop over the remaining edges in E_x and extract boundaries corresponding to vertices not visited before. By extracting reachable components first, we ensure that the corresponding reachable boundary of some unreachable component is always known before extraction, namely, the boundary containing the reachable endpoint of the considered edge in E_x. Therefore, the unreachable component is assigned to the unique border region that contains this reachable boundary. To make sure that we only compute actual border regions (in contrast to regions containing only accessible edges but no border edges), we can either check for border edges explicitly during traversal of the faces or disallow the traversal to start from an accessible edge.

Implementation Details. To extract the components of all border regions, we have to traverse faces of the planar input graph. This can be done using established data structures, such as doubly connected edge lists [Ber+08]. Instead, we propose a more cache-friendly data structure to represent the faces, which stores adjacent vertices of a face in contiguous memory. We use a single array that holds all faces of the graph. For each face, we store the sequence of vertices as they are found traversing the face in clockwise order starting at an arbitrary vertex. At the beginning and at the end of this sequence, we store sentinels that hold the index of the last and first entry of a vertex of the corresponding face, respectively. Traversing the face in either direction requires only a single scan along the array, jumping at most once to the beginning or end of the face. For consecutive vertices $u \in V_p$ and $v \in V_p$ in this array, we store at v its index in the corresponding twin face of the edge $\{u, v\}$. Finally, we store for every edge $\{u, v\}$ in the graph the two indices of the head vertex v in this data structure, i. e., its occurrence in the faces to the left and right of this edge.

To efficiently decide whether an edge is contained in E_x or if an edge in E_x was marked as visited, we store the set E_x as an array and sort it (e. g., by the index of the head vertex) before extracting the reachable components. Then, we can quickly retrieve an edge in E_x using binary search (we also tried using hash sets as an alternative approach, but this turned out to be slightly slower in preliminary experiments).

6.4 Range Polygons in Border Regions without Holes

Given a border region B with a reachable component R and a single unreachable component U, we present an algorithm for computing a polygon that separates R and U. In Section 6.5, we generalize our approach to the case $|U| > 1$.

Our approach adds an arbitrary border edge $e \in E_x$ to B, which connects both components R and U. Since we presume that G is strongly connected, such a border edge always exists. In the resulting hole-free, non-crossing polygon B', we compute a polygonal path with minimum number of segments that connects both sides of e. The algorithm of Suri [Sur86] computes such a *minimum-link path* π' in linear time. We obtain a separating polygon S' by connecting the endpoints of π' along e. It is easy to see that this yields a polygon with at most two additional segments compared to an optimal solution, as shown by Lemma 6.1 and illustrated in Figure 6.9.

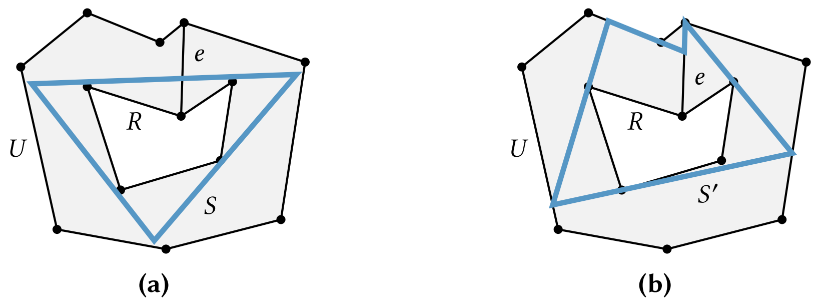


Figure 6.9: Separating polygons in a (shaded) border region. (a) The polygon S (blue) separating R and U with OPT = 3 links induces a path with four segments connecting both sides of e. (b) The polygon S' (blue) induced by a minimum-link path from e has OPT + 2 = 5 segments.

Lemma 6.1. *Let S be a polygon that separates R and U with minimum number of segments, and let* OPT *denote this number. Then S' has at most* OPT + 2 *segments.*

Proof. We can split S at e into a path π connecting both sides of e. Clearly, π has at most OPT + 1 links (if a segment of S crosses e, we split it into two segments with endpoints in e to obtain a path with OPT + 1 links). Since π' is a minimum-link path, we have $|\pi'| \leq |\pi| = \text{OPT} + 1$. Moreover, S' is obtained by adding a single subsegment of e to π', so its complexity is bounded by OPT + 2. □

We now address the subproblem of computing a minimum-link path between two edges of a simple polygon. The linear-time algorithm of Suri [Sur86] starts by triangulating the input polygon. To save running time for queries in our scenario, we triangulate all faces of the planar graph G_p during preprocessing. Afterwards, in each step of Suri's algorithm, a *window* (which we formally define in a moment) is computed. To obtain the windows in linear time, it relies on several calls to a subroutine computing visibility polygons. While this is sufficient to prove linear running time, it seems wasteful from a practical point of view. In the following, we establish important properties of windows (Section 6.4.1). Based on these, we present an alternative algorithm for computing the windows that also results in linear running time, but is much simpler (Section 6.4.2). It can be seen as a generalization of an algorithm by Imai and Iri [II87] for approximating piecewise linear functions.

In all what follows, when we consider the dual graph of a triangulation of a border region, we presume that vertices in the dual graph are connected by an edge if and only if their corresponding faces share an edge of the *triangulation* (as opposed to an edge of the border region). Moreover, we assume for the sake of simplicity that all points are in *general position*, i. e., there is no line that contains more than two points of a given set. Nevertheless, we also mention how our implementation handles such cases (which can occur in real-world instances).

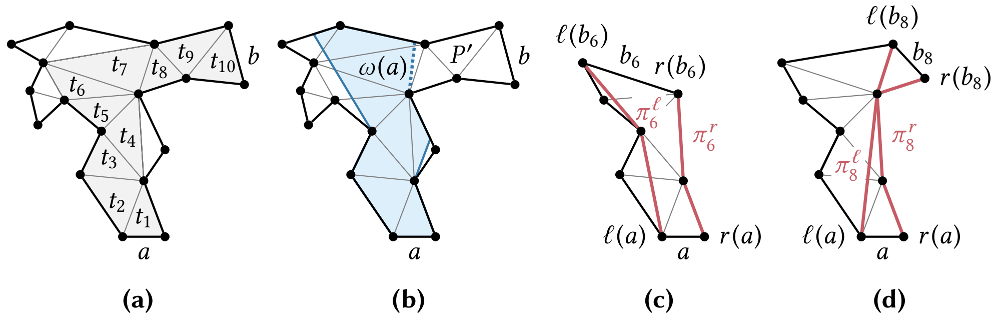


Figure 6.10: Important triangles, visibility, and shortest paths. (a) The (shaded) important triangles of a polygon with respect to the indicated edges a and b. (b) The window $\omega(a)$ is the dotted edge of the (shaded) visibility polygon. (c) The left and right shortest path (red) do not intersect for $i = 6$. (d) The shortest paths (red) have an intersection for $i = 8$.

6.4.1 Windows and Visibility

Let P be a simple polygon and let a and b be two edges of P. We want to compute a minimum-link polygonal path starting at a and ending at b that lies inside P. Let T be a graph obtained by arbitrarily triangulating P. Let t_a and t_b be the triangles containing a and b, respectively. As T is outerplanar, its (weak) dual graph has a unique path $t_a = t_1, t_2, \ldots, t_{k-1}, t_k = t_b$ from t_a to t_b; see Figure 6.10a. We call the triangles on this path *important* and their positions in the path their *indices*.

The *visibility polygon* $V(a)$ of the edge a in P is the polygon that contains all points that are visible from a. More formally, $V(a)$ contains a point $p \in \mathbb{R}^2$ in its interior if and only if there is a point q on a such that the line segment pq lies inside P. Let $i \in \{1, \ldots, k-1\}$ be the highest index such that the intersection of the triangle t_i with the visibility polygon $V(a)$ is not empty. The *window* $\omega(a)$ is the edge of $V(a)$ that intersects t_i closest (with respect to minimum Euclidean distance) to the edge between t_i and t_{i+1}; see Figure 6.10b. Note that $\omega(a)$ separates the polygon P into two parts. Let P' be the part containing the edge b that we want to reach. A minimum-link path from a to b in P is then obtained by adding an edge from a to $\omega(a)$ to a minimum-link path from $\omega(a)$ to b in P'. Thus, the next window is computed in P' starting with the previous window $\omega(a)$. In what follows, we show how $\omega(a)$ can be computed.

For some $i \in \{1, \ldots, k-1\}$, let T_i be the subgraph of T induced by the triangles $t_1, \ldots, t_i$ and let P_i be the polygon bounding the outer face of T_i. The polygon P_i has two special edges, namely a and the edge shared by t_i and t_{i+1}, which we call b_i. Let $\ell(a)$, $r(a)$, $\ell(b_i)$, and $r(b_i)$ be the endpoints of a and b_i, respectively, such that their clockwise order is $r(a)$, $\ell(a)$, $\ell(b_i)$, and $r(b_i)$ (think of $\ell(\cdot)$ and $r(\cdot)$ as being the respective left and right endpoints); see Figure 6.10c. We define the *left shortest path* π_i^ℓ to be the shortest polygonal path (shortest in terms of Euclidean length) that connects

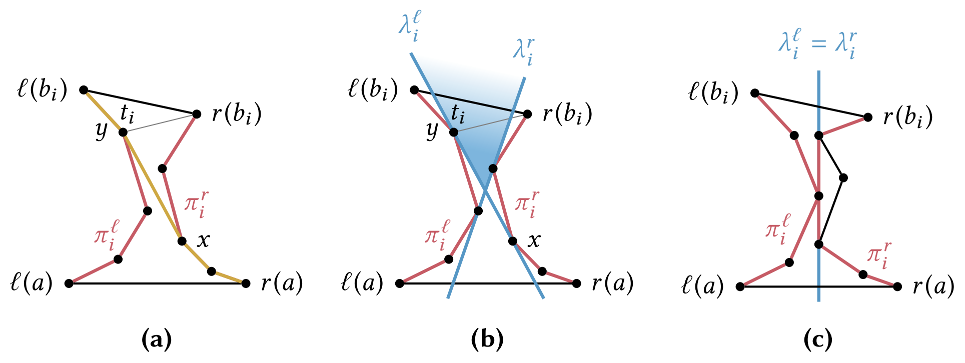


Figure 6.11: The visibility cone of an hourglass. (a) The shortest path (yellow) from $r(a)$ to $\ell(b_i)$ consists of a prefix of π_i^r, the segment xy, and a suffix of π_i^ℓ. (b) The two visibility lines λ_i^ℓ and λ_i^r (blue) spanning the (shaded) visibility cone. (c) A degenerate case, in which the visibility cone collapses to a line (as points are *not* in general position).

$\ell(a)$ with $\ell(b_i)$ and lies inside or on the boundary of P_i. The *right shortest path* π_i^r is defined analogously for $r(a)$ and $r(b_i)$; see Figure 6.10c.

Assume that the edge b_i is visible from a, i. e., there exists a line segment in the interior of P_i that starts at a and ends at b_i. Such a line segment separates the polygon into a left and a right part. Observe that it follows from the triangle inequality that the left shortest path π_i^ℓ and the right shortest path π_i^r lie inside the left and right part, respectively. Thus, these two paths do not intersect. Moreover, the two shortest paths are *outward convex* in the sense that the left shortest path π_i^ℓ has only left bends when traversing it from $\ell(a)$ to $\ell(b_i)$; see Figure 6.10c. The symmetric property holds for π_i^r. We note that the outward convex paths are sometimes also called *inward convex* and the polygon consisting of the two outward convex paths together with the edges a and b_i is also called *hourglass* [GH89]. The following lemma, which is similar to a statement shown by Guibas et al. [Gui+87, Lemma 3.1], summarizes the above observation. On the other hand, shortest paths that intersect are not outward convex in general; see Figure 6.10d.

Lemma 6.2. *If the triangle t_i is visible from the edge a (i. e., there is a line segment in the interior of P_i that starts at a and ends at a point in t_i), then the left and right shortest path in P_{i-1} have empty intersection. Moreover, if in fact these paths do not intersect, they are outward convex.*

Guibas et al. [Gui+87] argue that the converse of the first statement is also true, i. e., if the two paths have empty intersection, then the triangle t_{i+1} is visible from a. Their main arguments go as follows. The shortest path (with respect to Euclidean length) in the hourglass that connects $r(a)$ with $\ell(b_i)$ is the concatenation of a prefix of π_i^r, a line segment from a vertex x of π_i^r to a vertex y of π_i^ℓ, and a suffix of π_i^ℓ; see

Figure 6.11a. We call the straight line through x and y the *left visibility line* and denote it by λ_i^ℓ. We assume λ_i^ℓ to be oriented from x to y and call the vertices x and y the *source* and *target* of λ_i^ℓ, respectively. Analogously, one can define the *right visibility line* λ_i^r; see Figure 6.11b. We call the intersection of the half-plane to the right of λ_i^ℓ with the half-plane to the left of λ_i^r the *visibility cone*. It follows that the intersection of the visibility cone with the edge b_i is not empty and a point on the edge b_i is visible from a if and only if it lies in this intersection [Gui+87]. This directly extends to the following lemma.

Lemma 6.3. *If the left and right shortest path in P_{i-1} have empty intersection, t_i is visible from a. A point in t_i is visible from a if and only if it lies in the visibility cone.*

If π_i^r and π_i^ℓ are both outward convex and intersect, the visibility cone degenerates to a line; see Figure 6.11c. As mentioned above, we presume that all points are in general position, which prevents this special case. (In practice, it is handled implicitly by the implementation of line 7 in Figure 6.12 in the next section.)

The above observations justify the following approach for computing a window. We iteratively increase $i \in \{1, \dots, k\}$ until the left and the right shortest path of the polygon P_i intersect. We then know that the triangle t_{i+1} (or the edge b in case $i = k$) is no longer visible; see Lemma 6.2. Moreover, as the shortest paths did not intersect in P_{i-1}, the triangle t_i is visible from a; see Lemma 6.3. To find the window, we have to determine the edge of the visibility polygon $V(a)$ that intersects t_i closest to the edge between t_i and t_{i+1}. By the second statement of Lemma 6.3, the window must be a segment of one of the two visibility lines. Below, we discuss our algorithm in detail.

6.4.2 Fast Computation of Minimum-Link Paths

We describe our new algorithm to compute minimum-link paths, which we refer to as *FMLP (Fast Minimum-Link Paths)*. In particular, we detail the computation of the first window sketched in the previous section and describe what has to be done in later steps, when we start at a window instead of an edge. Furthermore, we argue that the algorithm runs in overall linear time. As before, we assume that we are given a polygon P, two edges a and b of P, and a triangulation T of P. The algorithm starts by computing the (unique) sequence $t_a = t_1, t_2, \dots, t_{k-1}, t_k = t_b$ of triangles connecting the triangles t_a and t_b containing a and b, respectively. Afterwards, it maintains shortest paths and visibility lines as defined in Section 6.4.1.

Computing the First Window. The algorithm starts from the edge a; see also Figure 6.12. Assume that the triangle t_i, with $i \in \{1, \dots, k-1\}$, is still visible from a, i. e., π_{i-1}^ℓ and π_{i-1}^r do not intersect. Assume further that we have computed the left and right shortest path π_{i-1}^ℓ and π_{i-1}^r as well as the corresponding visibility lines λ_{i-1}^ℓ and λ_{i-1}^r in a previous step. Assume without loss of generality that the three corners of the

```
// initial paths (as one-vertex sequences) and visibility lines
1  $\pi^\ell \longleftarrow [\ell(a)]$
2  $\pi^r \longleftarrow [r(a)]$
3  $\lambda^\ell \longleftarrow \text{line}(r(a), \ell(a))$
4  $\lambda^r \longleftarrow \text{line}(\ell(a), r(a))$
// run main loop
5  for $i \longleftarrow 1$ to $k$ do
6      if $r(b_i) = r(b_{i-1})$ then
           // if $b_i$ is not visible, return window
7          if $\ell(b_i)$ lies to the right of $\lambda^r$ then
8              $x \longleftarrow$ first intersection of $\lambda^r$ with $P$ after target($\lambda^r$)
9              return segment(target($\lambda^r$), $x$)
           // extend left path $\pi^\ell$ like in Graham's scan
10         append $\ell(b_i)$ to $\pi^\ell$
11         while last bend of $\pi^\ell$ is a right bend do
12             remove second to last element from $\pi^\ell$
           // if $\ell(b_i)$ lies in visibility cone, update left visibility line $\lambda^\ell$
13         if $\ell(b_i)$ lies to the right of $\lambda^\ell$ then
14             target($\lambda^\ell$) $\longleftarrow \ell(b_i)$
15             while $\lambda^\ell$ is not a tangent of $\pi^r$ at source($\lambda^\ell$) do
16                 source($\lambda^\ell$) $\longleftarrow$ successor of source($\lambda^\ell$) in $\pi^r$
17     else
           // case $\ell(b_i) = \ell(b_{i-1})$ is symmetric to the case $r(b_i) = r(b_{i-1})$
```

Figure 6.12: Pseudocode of FMLP (computation of the first window). Given a polygon P, an edge a in P, and a triangulation T of P, the algorithm computes the window $\omega(a)$.

triangle t_i are $\ell(b_{i-1})$, $\ell(b_i)$, and $r(b_i) = r(b_{i-1})$. There are three possibilities as shown in Figure 6.13, i. e., the new vertex $\ell(b_i)$ lies either in the visibility cone spanned by λ^ℓ_{i-1} and λ^r_{i-1} (Figure 6.13a), to the left of the left visibility line λ^ℓ_{i-1} (Figure 6.13c), or to the right of the right visibility line λ^r_{i-1} (Figure 6.13e).

By Lemma 6.3, a point in t_i is visible from a if and only if it lies inside the visibility cone. Thus, the edge b_i between t_i and t_{i+1} is no longer visible if and only if the new vertex $\ell(b_i)$ lies to the right of λ^r_{i-1}; see Figure 6.13e. In this case, we can stop and the desired window $\omega(a)$ is the segment of λ^r_{i-1} starting at its touching point with π^r_{i-1} and ending at its first intersection with an edge of P; see lines 7–9 of Figure 6.12 and Figure 6.13f. In the other two cases (Figure 6.13a and Figure 6.13c), we have to compute the new left and right shortest path π^ℓ_i and π^r_i, as well as the new visibility lines λ^ℓ_i and λ^r_i (Figure 6.13b and Figure 6.13d). Note that the old and new right shortest path

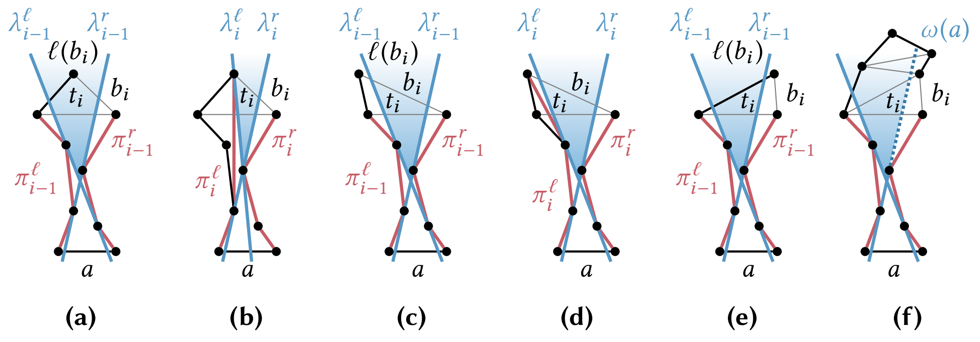


Figure 6.13: Visiting a new triangle. (a) The new vertex $\ell(b_i)$ lies in the visibility cone. (b) The updated left shortest path π^ℓ_i and left visibility line λ^ℓ_i. (c) The vertex $\ell(b_i)$ lies to the left of λ^ℓ_{i-1}. (d) The left shortest path has to be updated, the left visibility line remains unchanged. (e) The vertex $\ell(b_i)$ lies to the right of λ^r_{i-1}, i. e., t_{i+1} is not visible from a. (f) The dotted window $\omega(a)$ is a segment of λ^r_{i-1}.

π^r_{i-1} and π^r_i connect the same endpoints $r(a)$ and $r(b_{i-1}) = r(b_i)$. As the path cannot become shorter by going through the new triangle t_i, we have $\pi^r_i = \pi^r_{i-1}$. The same argument shows that $\lambda^r_i = \lambda^r_{i-1}$ (recall that the visibility lines were defined using a shortest path from $\ell(a)$ to $r(b_{i-1}) = r(b_i)$).

We compute the new left shortest path π^ℓ_i as follows; see lines 10–12 in Figure 6.12. Let x be the latest vertex on π^ℓ_{i-1} such that the prefix of π^ℓ_{i-1} ending at x concatenated with the segment from x to $\ell(b_i)$ is outward convex. We claim that π^ℓ_i is the path obtained by this concatenation, i. e., this path lies inside P_i and there exists no shorter path lying inside P_i. It follows by the outward convexity that there cannot be a shorter path inside P_i from $\ell(a)$ to $\ell(b_i)$. Moreover, by the assumption that π^ℓ_{i-1} was the correct left shortest path in P_{i-1}, the subpath from $\ell(a)$ to x lies inside P_i. Assume for the sake of contradiction that the new segment from x to $\ell(b_i)$ does not lie entirely inside P_i. Then it has to intersect the right shortest path and it follows that the right shortest path and the correct left shortest path have non-empty intersection, which is not true by Lemma 6.2.

To get the new left visibility line λ^ℓ_i, we have to consider the shortest path in P_i that connects $r(a)$ with $\ell(b_i)$. Let x and y be the source and target of λ^ℓ_{i-1}, respectively, i. e., the shortest path from $r(a)$ to $\ell(b_{i-1})$ is as shown in Figure 6.14a. If the new vertex $\ell(b_i)$ lies to the left of λ^ℓ_{i-1} (Figure 6.14b), then the shortest path from $r(a)$ to $\ell(b_i)$ also includes the segment from x to y. Thus, $\lambda^\ell_i = \lambda^\ell_{i-1}$ holds in this case. Assume the new vertex $\ell(b_i)$ lies to the right of λ^ℓ_{i-1} (Figure 6.14c). Let x' be the latest vertex on the path π^r_i such that the concatenation of the subpath from $r(a)$ to x' with the segment from x' to the new vertex $\ell(b_i)$ is outward convex in the sense that it has only right bends; see Figure 6.14c. We claim that this path lies inside P_i and that there is no shorter path inside P_i. Moreover, we claim that x' is either a successor of x in π^r_{i-1}

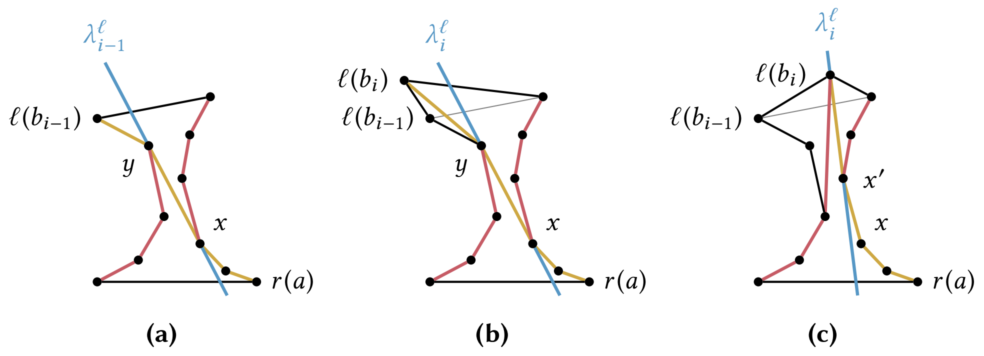


Figure 6.14: Updating the visibility line. (a) The shortest path from $r(a)$ to $\ell(b_{i-1})$ (yellow) defining the left visibility line λ_{i-1}^{ℓ}. (b) The visibility line does not change if $\ell(b_i)$ lies to the left of λ_{i-1}^{ℓ}. (c) Illustration of how the visibility line changes if $\ell(b_i)$ lies to the right of λ_{i-1}^{ℓ}.

or $x' = x$. Clearly, the concatenation of the path from $r(a)$ to x with the segment from x to $\ell(b_i)$ is outward convex, thus the latter claim follows. It follows that the segment from x' to $\ell(b_i)$ lies to the right of the old visibility line λ_{i-1}^{ℓ}. Thus, it cannot intersect the path π_i^{ℓ} (except in its endpoint $\ell(b_i)$), as π_{i-1}^{ℓ} lies to the left of λ_{i-1}^{ℓ}. Moreover, as we chose x' to be the last vertex on π_{i-1}^{r} with the above property, this new segment does not intersect π_i^{r} (except in x'). Hence, the segment from x' to $\ell(b_i)$ lies inside P_i. As before, it follows from the convexity that there is no shorter path inside P_i. Thus, λ_i^{ℓ} is the line through x' and $\ell(b_i)$, i. e., x' is the new source and $\ell(b_i)$ is the new target. Hence, lines 13–16 correctly compute the new left visibility line. Lemma 6.4 proves that $\omega(a)$ is also computed in linear time.

Lemma 6.4. *Let t_h be the triangle with the highest index $h \in \{1, \ldots, k\}$ that is visible from a. Then FMLP computes the first window $\omega(a)$ in $O(h)$ time.*

Proof. We already argued that FMLP correctly computes the first window. To show that it runs in $O(h)$ time, first note that the polygon P_h has linear size in h. Thus, it suffices to argue that the running time is linear in the size of P_h. In each step i, with $i \in \{1, \ldots, k\}$, we first check whether the next triangle is still visible by testing whether the new vertex $\ell(b_i)$ (or $r(b_i)$) lies to the right of the visibility line λ_{i-1}^{r} (or to the left of λ_{i-1}^{ℓ}). This takes only constant time. When updating the left and right shortest path, we have to iteratively remove the last vertex of the previous path until the resulting path is outward convex. This takes linear time in the number of vertices we remove. However, a vertex removed in this way can never be part of a left or right shortest path again. Thus, the number of these removal operations over all h steps is bounded by the size of P_h. When updating the visibility lines, the only operation potentially consuming more than constant time is finding the new source x'. As x' is a successor of the previous source x (or $x' = x$), we never visit a vertex of P_h twice in

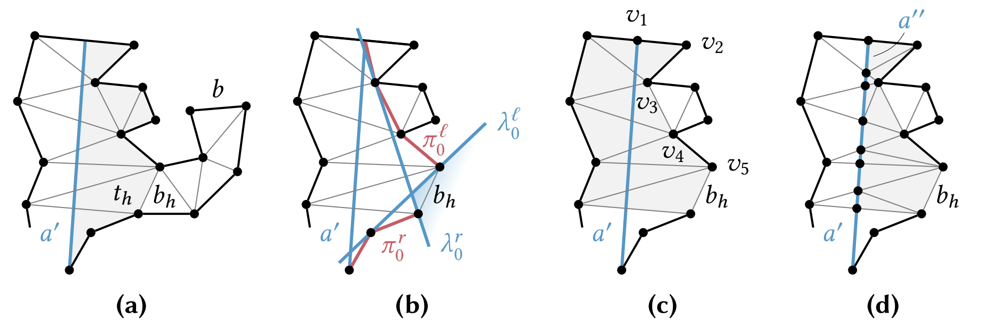


Figure 6.15: Initializing computation of the second window. (a) The polygon P' we are interested in after computing the first window a'. The initial part P'_0 is shaded. (b) Initial left and right shortest path π_0^ℓ and π_0^r (red) with corresponding visibility lines λ_0^ℓ and λ_0^r (blue). (c) The sequence $v_1, \dots, v_5$ we use for Graham's scan. Triangles of P intersected by a' are shaded. (d) Computing the shortest path from $\ell(a'') = \ell(a')$ to $\ell(b_h)$ in the subdivided polygon (shaded) using the algorithm from Figure 6.12 actually applies Graham's scan to $v_1, \dots, v_5$.

this type of operation. Thus, the total running time of finding these successors over all h steps is again linear in the size of P_h. □

Initialization for Subsequent Windows. As mentioned before, the first window $\omega(a)$ we compute separates P into two smaller polygons. Let P' be the part including the edge b (and not a). In the following, we denote $\omega(a)$ by a'. To get the next window $\omega(a')$, we have to apply the above procedure to P' starting with a'. However, this would require us to partially retriangulate the polygon P'. More precisely, let t_h be the triangle with the highest index $h \in \{1, \dots, k-1\}$ that is visible from a and let b_h be the edge between t_h and t_{h+1}; see Figure 6.15a. Then b_h separates P' into an initial part P'_0 (the shaded part in Figure 6.15a) and the rest (having b on its boundary). The latter part is properly triangulated, however, the initial part P'_0 is not. The conceptually simplest solution is to retriangulate P'_0. However, this would require an efficient subroutine for triangulation (and dynamic data structures that allow us to update P and T, which produces overhead in practice). Instead, we propose a much simpler method for computing the next window.

The general idea is to compute the shortest paths in P'_0 from $\ell(a')$ to $\ell(b_h)$ and from $r(a')$ to $r(b_h)$; see Figure 6.15b. We denote these paths by π_0^ℓ and π_0^r, respectively. Moreover, we want to compute the corresponding visibility lines λ_0^ℓ and λ_0^r. Afterwards, we can continue with the correctly triangulated part as in Figure 6.12.

Concerning the shortest paths, we assume as before that the window $\omega(a) = a'$ is a segment of the right visibility line λ_h^r; the other case is symmetric. First, note that the right shortest path π_0^r is a suffix of the previous right shortest path, which we already know. For the left shortest path π_0^ℓ, consider the polygon induced by the

```
    // initialization
1   $a' \longleftarrow$ previous window
2   $\pi_0^\ell \longleftarrow \emptyset$
3   $\pi^r \longleftarrow$ right shortest path computed in the previous step
4   $\pi_0^r \longleftarrow$ suffix of $\pi^r$ starting with $r(a')$
    // apply Graham's scan to the sequence $v_1, \dots, v_g$
5   for $i \longleftarrow 1$ to $g$ do
6       append $v_i$ to $\pi_0^\ell$
7       while last bend of $\pi_0^\ell$ is a right bend do
8           remove second to last element from $\pi_0^\ell$
9   $\lambda_0^\ell \longleftarrow$ tangent of $\pi_0^r$ through $\ell(b_h)$
10  $\lambda_0^r \longleftarrow$ tangent of $\pi_0^\ell$ through $r(b_h)$
11  return $(\pi_0^\ell, \pi_0^r, \lambda_0^\ell, \lambda_0^r)$
```

Figure 6.16: Pseudocode of FMLP (initialization of subsequent windows). After computing the previous window a', this step computes the new initial left and right shortest paths with corresponding visibility lines.

triangles that are intersected by a'; see Figure 6.15c. Let $[v_1, \dots, v_g]$ be the path on the outer face of this polygon (in clockwise direction) from $\ell(a') = v_1$ to $\ell(b_h) = v_g$. We obtain π_0^ℓ using *Graham's scan* [Gra72] on the sequence $v_1, \dots, v_g$, i. e., starting with an empty path, we iteratively append the next vertex of the sequence $v_1, \dots, v_g$ while maintaining the path's outward convexity by successively removing the second to last vertex if necessary; see Figure 6.16 for pseudocode. Note that applying Graham's scan to arbitrary sequences of vertices may result in self-intersecting paths [Byk78]. Below, Lemma 6.5 proves that this does not happen in our case.

It remains to compute the visibility lines λ_0^ℓ and λ_0^r corresponding to the hourglass consisting of a', b_h, and the shortest paths π_0^ℓ and π_0^r. Note that the whole edge b_h is visible from a', since a' intersects the triangle t_h. Thus, the visibility lines go through the endpoints of b_h. It follows that λ_0^ℓ is the line that goes through $\ell(b_h)$ and the unique vertex on π_0^r such that it is tangent to π_0^r; see Figure 6.15b. Clearly, this can be found in linear time in the length of π_0^r. The same holds for the right visibility line.

Lemma 6.5. *The FMLP algorithm computes the initial left and right shortest paths π_0^ℓ and π_0^r, as well as the corresponding visibility lines λ_0^ℓ and λ_0^r in $O(|P_0'|)$ time.*

Proof. We mainly have to prove that the path π_0^ℓ obtained after applying Graham's scan on the sequence $v_1, \dots, v_g$ actually is the shortest path from $\ell(a)$ to $\ell(b_h)$ in P_0' (which includes that it is not self-intersecting). This can be seen by using arguments we made for computing the first window. To this end, we reuse the triangulation we have for P by placing new vertices where a' crosses triangulation edges; see Figure 6.15d. Note

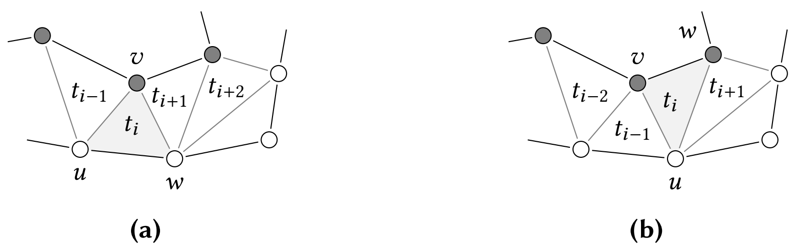


Figure 6.17: Identifying the next important triangle. (a) The next important triangle t_{i+1} shares with t_i the unique edge vw that is not uv and has exactly one reachable endpoint. (b) The next important triangle t_{i+1} contains the edge uw.

that the resulting polygon, which we denote by P_0'', is almost triangulated, namely, each face is a triangle or a quadrangle. Thus, we can triangulate P_0'' by adding one new edge in each quadrangle as in Figure 6.15d. Note that a' is separated into several edges in P_0''; let a'' be the topmost of these edges (i. e., the last one in clockwise order). Assume we want to compute the minimum-link path from a'' to b_h in P_0''. First, note that the triangle t_h is visible from a''. Thus, our algorithm for computing the first window computes the shortest path from $\ell(a') = \ell(a'')$ to $\ell(b_h)$. Note further that the vertices visited in lines 10–12 of the algorithm outlined in Figure 6.12 are the vertices $v_1, \dots, v_g$, in this order. Thus, the algorithm shown in Figure 6.12 actually constructs the left shortest path by using Graham's scan on the sequence $v_1, \dots, v_g$. It follows that directly applying Graham's scan to the sequence $v_1, \dots, v_g$ correctly computes the left shortest path in P_0' from $\ell(a')$ to $\ell(b_h)$. Furthermore, the running time of the algorithm in Figure 6.16 is linear in the size of P_0'. □

We compute subsequent windows as described above, until the last edge b is found. The actual minimum-link path π' is obtained by connecting each window $\omega(a)$ to its corresponding first edge a with a straight line [Sur86]. Linear running time of the algorithm follows immediately from Lemma 6.4 and Lemma 6.5. Theorem 6.6 summarizes our findings.

Theorem 6.6. *Given two edges a and b of a simple polygon P, the FMLP algorithm computes a minimum-link path from a to b contained in P in linear time.*

Implementation Details. To obtain the desired polygon that separates R and U, we can connect the first and last segment of π' along the initial border edge $e \in E_x$, as described above. However, to potentially save a segment and for aesthetic reasons, we first test whether the last window can be extended to intersect the first segment of the path without intersecting the boundary of R or U (observe that this is the case in Figure 6.9b). Thus, we continue the computation of the last window from t_0 after the edge b was found.

We do not construct P and its triangulation T explicitly, but work directly on the triangulated input graph. The next important triangle is then computed on-the-fly as follows. Consider an important triangle $t_i = uvw$, where $i \in \{1, \ldots, k-1\}$, and let uv be the edge shared by t_i and t_{i+1}; see Figure 6.17. Clearly, exactly one endpoint of uv is part of the reachable boundary, so without loss of generality let u be this endpoint. Then the next important triangle is the triangle sharing vw with t_i if w is reachable, and the triangle sharing uw with t_i otherwise. In other words, the next triangle is determined by the unique edge that has exactly one reachable endpoint. Faces of the triangulated graph are stored in a single array, similar to the data structure for the planar graph described in Section 6.3.1 (we do not use sentinels, though, since triangular faces have constant size). Our implementation of FMLP operates on this data structure to determine the important triangles.

6.5 Heuristic Approaches for General Border Regions

A border region B may consist of multiple unreachable components, i. e., we may have $|U| > 1$, whereas $|R| = 1$ always holds. In this general case, it is not clear whether one can compute a range polygon of minimum complexity (without self-crossings) that separates R and U in polynomial time [Gui+93]. Even for the simpler subproblem of computing a minimum-link path in a polygon with several components (without assigning them to the reachable or unreachable boundary), the fastest known algorithm has quadratic running time [Kos+16, MRW92]. This is impractical for large instances. In fact, the problem was recently shown to be 3SUM-hard [MPS14], so algorithms with subquadratic running time may not even exist [GO95]. Therefore, we propose four heuristic approaches with (almost) linear running time (in the size of B) that are simple and fast in practice. Figure 6.18 shows example outputs of the different heuristics.

Given a border region B, the first approach (Section 6.5.1) simply computes and returns the reachable boundary R; see Figure 6.18a. This results in a polygon that resembles the output of known algorithms for isochrones [MG10]. Since the complexity of the range polygons can become quite high, we propose more sophisticated heuristics. Our second approach (Section 6.5.2) uses the triangulation of the input graph to subdivide B using edges for which either both endpoints are in R or both endpoints are in U. The modified instances consist of single unreachable components, which are separated from the reachable component by the algorithm from Section 6.4; see also Figure 6.18b. The third algorithm (Section 6.5.3) inserts new edges into B in order to connect the components of U and thereby create an instance with $|U| = 1$. Afterwards, we compute a minimum-link path in the resulting border region as in Section 6.4; see also Figure 6.18c. Our fourth heuristic (Section 6.5.4) modifies the FMLP algorithm to compute a possibly self-intersecting minimum-link path separating R and U with minimum number of segments; see Figure 6.18d. Consequently, the

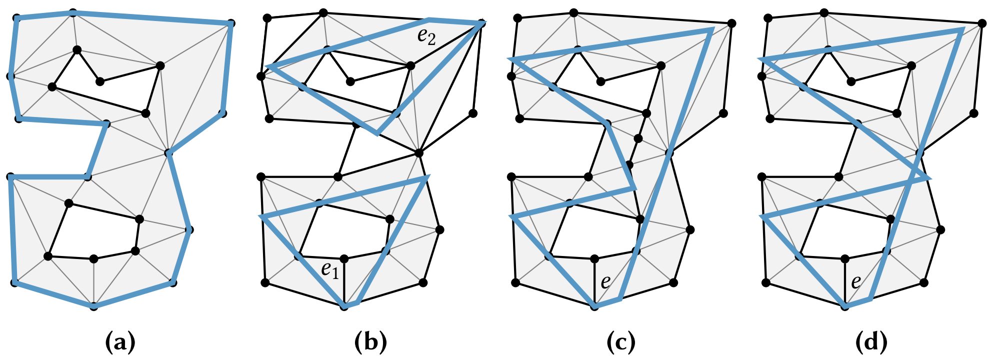


Figure 6.18: Example output of different heuristic approaches, for a (shaded) border region with two unreachable components. Minimum-link paths are computed from indicated border edges. (a) Output of the approach presented in Section 6.5.1, which returns the reachable boundary. (b) The approach in Section 6.5.2 subdivides border regions into smaller subinstances. (c) The approach in Section 6.5.3 connects unreachable components to obtain a simpler instance. (d) Our last approach in Section 6.5.4 computes a polygon whose edges may cross each other.

resulting polygon has at most two more segments than an optimal solution. We rearrange it at points where its edges cross each other to obtain a new range polygon without self-intersections.

6.5.1 Extracting the Reachable Component

Given a border region B, the first approach returns the reachable boundary R. This results in a range polygon that is similar to known approaches, which essentially consist of extracting the reachable subgraph [GBI12, MG10]. Note that this approach does not have to compute the unreachable boundary explicitly. Thus, we can improve performance by modifying the extraction of border regions described in Section 6.3.2, such that only the reachable part of the boundary is traversed. In a sense, this extraction algorithm can be seen as an efficient implementation of previous approaches [GBI12, MG10]. Its linear running time (in the size of B) follows from the fact that we traverse every edge of R once, and every boundary edge or accessible edge of the planar graph G_p contained in B a constant number of times. Clearly, the number of these edges is linear in the size of B.

6.5.2 Separating Border Regions along their Triangulation

The idea of this approach is as follows. For each border region B, we add all edges of its triangulation that either connect two reachable vertices or two unreachable vertices of G_p to B, possibly splitting B into multiple regions B' (see edges separating the border region in Figure 6.18b). For each region B', we obtain an unreachable boundary U'

with $|U'| \leq 1$, since any pair of components in U must be connected by an edge of the triangulation or separated by an edge with two endpoints in R. Thus, we run the algorithm presented in Section 6.4 on each instance B' with $|U'| = 1$ to get the range polygon. Linear running time follows, as we run the linear-time FMLP algorithm on disjoint subregions of B. Clearly, the number of edges we add to B is not minimal, i. e., in general we could omit some edges and still obtain $|U'| \leq 1$ for each region B'. On the other hand, computing the set of separating edges described above is trivial, making our approach very simple.

We describe how the heuristic is implemented without explicitly constructing the border region B. Instead, we use the set E_x to identify the border regions that need to be handled (recall that E_x contains the border edges and accessible edges of all border regions). We loop over all edges in this set and check for each border edge $\{u, v\} \in E_x$ whether it was already visited. If this is not the case, we run FMLP from the triangle to the left of this edge (accessible edges in E_x are skipped by this loop, since they connect two reachable vertices and are thus considered part of the boundary). The sequence of important triangles is computed on-the-fly, as described in Section 6.4.2. Whenever the algorithm passes a border edge, it is marked as visited.

The heuristic can be seen as a simple but effective way of producing border regions with a single unreachable component. It is very easy to implement and even simplifies aspects of FMLP, because modified border regions contain only important triangles (all other triangles are removed from modified border regions). Thus, computational effort for finding the second endpoint of a new window and the initial visibility lines is restricted to a single triangle, enabling the use of simpler data structures. On the other hand, the output of the algorithm heavily depends on the triangulation of the input graph. In addition to that, the number of modified regions B' can become quite large (see Section 6.6.2). Therefore, we propose a more sophisticated way to obtain regions with a single unreachable component in Section 6.5.3.

6.5.3 Connecting Unreachable Components

Our next heuristic adds new edges to border regions with more than one unreachable component, such that they connect all unreachable components without intersecting the reachable boundary; see Figure 6.18c for an example. We obtain a modified instance B' with a single unreachable component and apply the algorithm from Section 6.4. Note that in general, unreachable components cannot be connected by straight lines; see Figure 6.19b. For a similar (more general) scenario, Guibas et al. [Gui+93] propose an approach to compute a subdivision that requires $O(h)$ more segments than an optimal solution (where h is the number of components in the input region). We propose a heuristic approach without any nontrivial guarantee. However, it is easy to implement and provides high-quality solutions in practice (see Section 6.6.2).

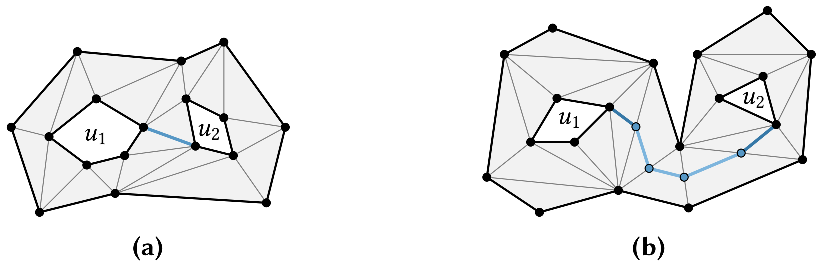


Figure 6.19: A border region with two unreachable components u_1 and u_2. (a) The indicated components u_1 and u_2 are connected by a single edge (blue) of the triangulation. (b) The unreachable components u_1 and u_2 are connected by two connecting edges (dark blue) and several bridging edges (light blue), which use the inserted dummy vertices (blue).

Given a border region B whose unreachable boundaries U consist of multiple components, our algorithm runs a BFS on the dual graph of the triangulation of B to find paths that connect the unreachable components. Then, we add new edges that connect the components and retriangulate the modified border region.

We distinguish two cases for connecting two unreachable components u_1 and u_2 in U. First, u_1 and u_2 may be connected by a single edge of the triangulation, i. e., an edge that has an endpoint in each component. Then, we can simply add this edge to B to connect u_1 and u_2; see Figure 6.19a. Second, we have to deal with components that are not connected by such an edge. In this case, there exists at least one edge e in the triangulation of B such that both endpoints of e are on the reachable boundary, hence e separates B into two subregions containing u_1 and u_2, respectively. Thus, any path connecting u_1 and u_2 in B crosses e. Our goal is to find a short path in the dual graph of the triangulation that connects u_1 and u_2. Then, we add new edges to the corresponding sequence of triangles to connect u_1 and u_2 in B; see Figure 6.19b. Afterwards, we locally retriangulate the modified part of B.

Connecting Components. Our algorithm starts by checking for each pair of components whether they are connected by a single edge in the triangulation of B. This can be done in a linear scan over the vertices of every unreachable component, checking for each vertex its outgoing edges in the triangulation. Whenever an edge is found that connects two unreachable components, we merge these components and consider them as the same component in the further course of the algorithm (making use of a union-find data structure [MS08]).

To connect all remaining unreachable components after this first step, we proceed as follows. Consider the (weak) dual graph of the triangulation of B. Since no pair of remaining unreachable components can be connected by a single edge in the primal graph, each triangle intersects at most one unreachable component. We assign a component to each vertex in the dual graph, namely, the reachable component if the

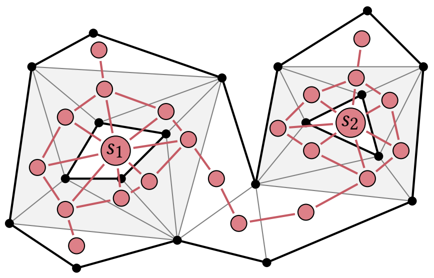


Figure 6.20: The dual graph (red) of a border region, with indicated super sources s_1 and s_2. Shaded triangles are assigned to one of the two unreachable components.

corresponding triangle contains only reachable vertices, and the unique unreachable component this triangle intersects otherwise. For each unreachable component, we add a super source to the dual graph that is connected to all vertices assigned to this component; see Figure 6.20. Our goal is to find a tree of minimum total length in this graph that connects all super sources, i. e., a minimum Steiner tree. Since this poses an $\mathcal{NP}$-hard problem in general [GJ79], we use the approach of Kou et al. [KMB81], which achieves an approximation ratio below 2. Its basic idea is to iteratively add shortest paths between two sources that are not connected yet in a greedy fashion. A search proposed by Wu et al. [WWW86] computes these paths in a graph with a given cost function. (Faster algorithms exist for this weighted case [Meh88, Wid87].) Since the (dual) graph has no edge costs in our case, the algorithm by Wu et al. boils down to a multi-source variant of a BFS, which we now describe in more detail.

Given an undirected graph $G = (V, E)$ and $k \in \{1, \dots, n\}$ source vertices, our search keeps vertex labels $\ell(v)$ for each $v \in V$ to mark visited vertices and store their parents in the search together with the corresponding source vertex of a path that reached the vertex. We use a union-find data structure (with k elements) to maintain connectivity of sources. Initially, we mark all sources as visited and set each as its own source. Moreover, all source vertices are inserted into a FIFO queue. Then, the main loop runs until all sources are connected. In each step, the search extracts the next vertex $u \in V$ from the queue and checks all incident edges $\{u, v\} \in E$. If v was not visited, the algorithm marks its label as visited and sets its source as the source of the label of u. Additionally, v is inserted into the queue. On the other hand, if v was already visited, we found a path that connects two sources. We check whether the sources of the labels $\ell(u)$ and $\ell(v)$ are not connected yet. If this is the case, we found the first path that connects them, so we unify the sources (i. e., they are considered equal in the further course of the algorithm). The actual path can be retrieved by backtracking from u and v, respectively, following parent pointers until the source is reached. The concatenation of both paths yields a path that connects two source vertices. The algorithm stops when all sources are connected.

After the search has terminated, we *split* some triangles by adding new vertices and edges to B in the following manner (see also Figure 6.19b). Consider a path in the dual graph of B connecting the super sources representing two components u_1 and u_2. We remove the first and last vertex of this path (because these are previously added super sources). For the remaining path, consider the corresponding sequence of triangles in B. Clearly, all but the first and last triangle of this sequence only have endpoints in the reachable component (otherwise, backtracking would have started or stopped earlier). For every edge shared by two triangles in the path, we add a *bridging vertex* at the center of this edge. Thus, a bridging vertex is always contained in an edge with two reachable endpoints. Between any pair of bridging vertices contained in the same triangle, we add a *bridging edge* connecting them. Finally, at the first and last triangle of the path, we add a *connecting edge* from the bridging vertex to the unique endpoint that belongs to an unreachable component. Assigning all added vertices and edges to the unreachable boundary, the resulting border region B' contains a connected unreachable component $U' \supseteq \{u_1, u_2\}$.

Finally, we add new edges (if necessary) to any created quadrangles to maintain the triangulation. Thus, the resulting modified border region B' is triangulated and its unreachable boundary consists of a single component. We run the algorithm described in Section 6.4 to obtain the desired range polygon.

For correctness, we need to show that the union of all edges added according to the computed Steiner tree creates no crossings. First, note that bridging edges in a triangle never cross each other. Second, we claim that if a connecting edge is inserted in some triangle, no other edge is added to that triangle. Since a connecting edge has an endpoint in some unreachable component, at most one edge of the triangle has two reachable endpoints. Hence, it contains at most one bridging vertex and no bridging edge. Moreover, the triangle contains at least one bridging vertex (the other endpoint of the connecting edge). Thus, it has exactly two reachable endpoints and cannot contain more than one connecting edge.

Complexity. The BFS described above visits each vertex of the dual graph at most once. In each step, a constant number of calls to the union-find data structure is made to check whether sources of two given labels are connected and unify them if necessary (vertex degree is constant except at super sources, where no checks are performed). All other operations require constant time. Using path compression for the union-find data structure [Tar75], this yields a running time of $O(n\alpha(n))$ of the BFS, where n is the number of vertices in the dual graph (which is linear in the size of B) and α the inverse Ackermann function. Since the remaining steps of the heuristic (adding vertices and edges to triangles, computing a minimum-link path) require linear time, the overall running time is almost linear.

Improvements. In practice, the performance of the BFS is dominated by the number of visited vertices. We propose tuning options that reduce this number significantly, without affecting correctness of the approach (but the output may change slightly).

One crucial observation is that realistic instances of border regions often have an unreachable boundary consisting of one large component (the major part of the unreachable subgraph), and many tiny components (e. g., unreachable dead ends in the road network), similar to Figure 6.5 in Section 6.1.2. Then, the search from the large component dominates running time. Instead, we can run the BFS starting from all but the largest component. This requires only negligible overhead (we identify the largest component in an additional linear scan), but searches from small components are likely to quickly converge to the large component. In preliminary experiments, this reduced running time significantly. Furthermore, after extracting the next vertex from the queue, we first check whether its source was connected to the largest component in the meantime. If this is the case, we prune the search at this vertex (i. e., we do not check its incident edges), because it now represents the search from the largest component. Similarly, before running the BFS, we omit vertices of the largest component when checking for edges in the triangulation that connect two components.

Going even further, we always expand the search from the component that is currently the smallest. In its basic variant, the BFS uses a queue to process vertices in FIFO order. For better (empirical) performance, we replace it with a priority queue whose elements are components (represented by source vertices). Additionally, we maintain a queue for each component, storing and extracting vertices in FIFO order. In the priority queue, each component uses its complexity (i. e., its number of edges) as key. In each step of the BFS, we check for the component with the smallest key in the priority queue and extract the next vertex from the queue of this component. If it has run empty, we remove the component from the priority queue. New vertices are always added to the queue that corresponds to the component of their source label. Whenever two components are unified, we also update them in the priority queue by removing one of the two involved components, attaching its queue to the other component, and updating the key accordingly. If components in the priority queue are implemented as *lists* of queues, new queues can be removed and reattached in constant time. In total, the use of a priority queue then increases the asymptotic running time of the BFS by a logarithmic factor, but we observe a significant speedup in practice.

Data Structures and Implementation Details. When running a BFS on the dual graph of a border region B, we implicitly represent the search graph using the triangulation of the planar input graph G_p. To determine incident edges of a vertex in the dual graph, we check the edges of its triangle in the primal graph. If such an edge is not present in G_p (i. e., it was added during triangulation) or contained in E_x, there is an edge in the dual graph connecting the triangle to the twin triangle of this edge.

In the improved variant that makes use of a priority queue, our implementation actually keeps a single queue per component, rather than a list of queues. Whenever some components are unified, the keys of affected components in the priority queue are updated in a scan over all elements it contains. This increases asymptotic running time of the BFS by another linear factor (in the number of components, which can be linear in the size of the border region). However, the number of components is usually small in practice and we avoid overhead for maintaining dynamic lists of queues.

To avoid costly reinitialization of vertex labels between queries, we make use of *timestamps* [Paj13], implicitly encoded within the component indices of labels to save space. After every search, the global timestamp is increased by the number of unreachable components in the border region. Before storing a component index in a label, the index is increased by the global timestamp. A stored label is invalid if its index is below the global timestamp. Otherwise, we subtract the global timestamp to retrieve the actual index of the valid label.

Backtracking runs on-the-fly during the BFS and stops whenever we reach a previously split triangle, since this means we have reached a previously computed path. We maintain flags at each triangle, to determine whether a bridging edge or a connecting edge should be inserted (and if so, between which pair of endpoints). We also build a list of all split triangles, for fast (sequential) access to all triangles that were split, in order to add the corresponding vertices and edges in the triangulation after the BFS has terminated. These additional edges and vertices are stored as temporary modifications in the triangulation of G_p, where we make use of the following data structures. Edges of the triangulation that are added to U' (to connect two unreachable components) are explicitly stored in a list. To quickly check whether some edge of the triangulation was added to U', we sort this list (e. g., by head vertex index) after the BFS terminated to enable binary search. In our setting (some 1 000 inserted edges for the hardest queries), this turned out to be slightly faster than using hash sets. To store bridging edges and connecting edges, we temporarily modify the triangulation. To this end, we add an invalidation flag and a temporary index to the vertices of every triangle in the triangulation of G_p. Moreover, we maintain a list of temporary triangles. Each vertex in a split triangle is marked as invalid and its temporary index is set to the corresponding entry in this list. Before retrieving a triangle vertex, we first check whether it is invalid and redirect to the temporary vertex if this is the case. For faster reinitialization, we replace invalidation flags by timestamps, similar to component timestamps described above.

Finally, when splitting triangles, we have to set the twins of all new edges. If the twin triangle was not created yet, we store the pending edge in a list. This list is searched for existing twins whenever a new triangle is added. If a twin is found, we set twins for both affected edges, and remove them from the set.

6.5.4 Computing Self-Intersecting Minimum-Link Paths

Our last approach computes a minimum-link path in B that separates the reachable boundary from the unreachable boundaries. While the resulting polygon has at most OPT + 2 segments, it may intersect itself; see Figure 6.18d. To obtain a range polygon from a self-intersecting polygon, we rearrange it accordingly at crossings.

We describe how minimum-link paths are computed in border regions with multiple unreachable components, by making modifications to the FMLP algorithm from Section 6.4. First, note that the (weak) dual graph of the triangulation of B is not outerplanar if $|U| > 1$. Consequently, paths between vertices in the dual graph are no longer unique. In fact, vertices may now occur multiple times in the path traversed by FMLP; see the corresponding sequence of triangles crossed by the polygon in Figure 6.18d. In what follows, we first show how the sequence of important triangles is obtained in this general case. Then, we describe modifications that are necessary to retain correctness of FMLP when running on this sequence of important triangles.

Computing Important Triangles. Given a border edge e of a border region B, we are interested in a minimum-link path that connects both sides of e and separates the reachable boundary from all unreachable boundaries. We compute a sequence $t_1, \dots, t_k$ of triangles such that any minimum-link path with the above property must pass the sequence in this order. Our approach runs in two phases. The first phase traverses the reachable boundary of B and lists all encountered border edges with respect to the triangulation, i. e., all edges with one endpoint in each R and U, even if they are not present in the input graph G_p. Clearly, the minimum-link path must intersect exactly these border edges in the same order to ensure that all unreachable vertices are on the same side of the path. By construction, any pair of consecutive border edges a and b in this list is connected by a path P in the dual graph that contains no border edge besides a and b (where an edge in the dual graph is called border edge if it corresponds to a border edge of the primal graph). In fact, observe that P is actually unique, since any cycle in the dual graph contains at least one border edge. The second phase computes this unique path P for each pair of consecutive border edges. The concatenation of all these paths yields the actual sequence of important triangles. It serves as input for a modified FMLP algorithm that computes a minimum-link path connecting both sides of the border edge e in B.

During the first phase, we exploit the fact that the reachable boundary of B is always connected. We assign *indices* to all edges in the triangulation that are contained in the border region and intersect the reachable boundary, according to the order in which they are traversed starting from e. In doing so, we distinguish both sides of edges; see Figure 6.21. For consistency, sides of edges that are not traversed get the index ∞. Clearly, this information can be retrieved in a single traversal of the reachable boundary, similar to the procedure described in Section 6.5.1, but running on the triangulation of the border region. During this traversal, we also collect an ordered list of indices corresponding to border edges. Observe that every border edge in B is traversed exactly once.

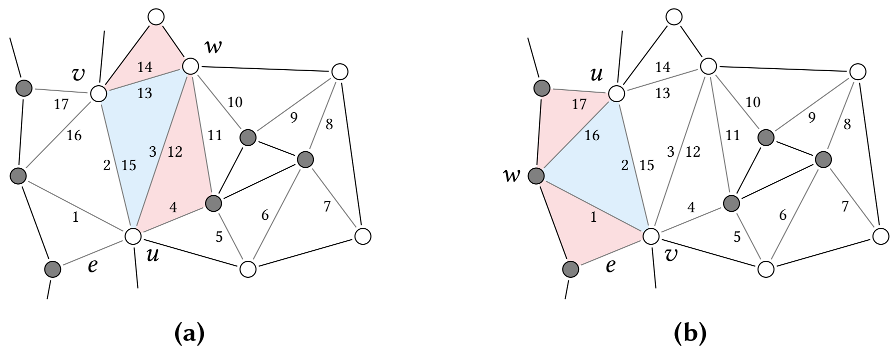


Figure 6.21: Border region with edge indices after starting traversal at the indicated edge e. Note that edges have two indices (one for each direction of traversal). Indices are ∞ if not specified. Indices of border edges are also stored in an array $[1, 4, 5, 6, 7, 8, 9, 10, 11, 16, 17]$. (a) The third visited triangle in the second phase (shaded blue) has two possible next triangles t_{uw} and t_{vw} (shaded red). The next triangle is t_{uw}, because the index of the edge vw with greater index (13) exceeds the next border edge index (4). (b) The next triangle in this example is t_{uw}. The index of the next border edge is updated from 16 to 17.

The second phase runs on the dual graph of the triangulation and retrieves the desired sequence of triangles. A key observation is that this sequence must pass all border edges exactly once and in increasing order of their indices. Therefore, we can compute the sequence of important triangles as follows. We maintain the index of the next border edge that was not traversed yet, initialized to the first element of the list. Starting at the triangle t_1 containing the first border edge e, we add triangles to the sequence of important triangles until e is reached again. Let $t_i = uvw$ denote the previous triangle that was appended to this sequence. We determine the next triangle t_{i+1} as follows; see Figure 6.21. Let uv be the unique edge shared by t_i and t_{i-1} (in the case of $i = 1$, we obtain $uv = e$). To determine the next triangle, we consider the two possible triangles t_{uw} containing the edge uw and t_{vw} containing vw. Without loss of generality, let the index of uw be smaller than the index of vw and thus, finite. This implies that uw is not contained in the boundary of B (otherwise, it would have index ∞). If both u and w are part of the reachable boundary, we know that uw separates B into two subregions; see Figure 6.21a. Thus, t_{uw} is the next triangle if and only if the subregion containing t_{uw} contains a border edge that was not passed yet. Therefore, we continue with t_{uw} if and only if the index of the other edge vw is greater than the index of the next border edge. If either u or w is part of an unreachable boundary, uw is the next border edge; see Figure 6.21b. We update the index of the next border edge to the next element in the according list.

We continue until the first edge e is reached again. Note that the second phase (traversing the dual graph) can be performed on-the-fly during minimum-link path computation (i. e., the sequence of triangles does not have to be built explicitly).

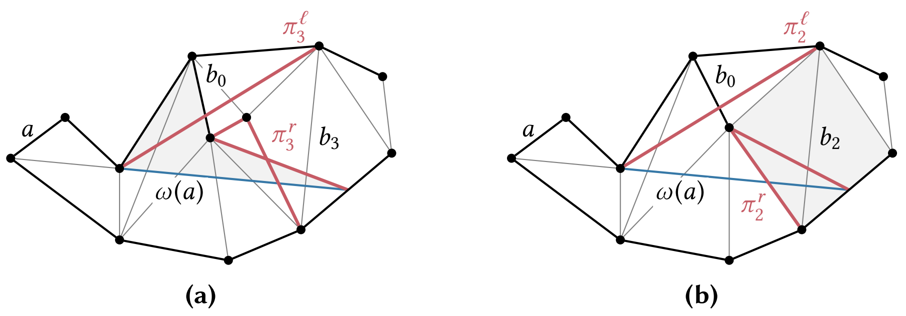


Figure 6.22: Shortest paths in general border regions. Assume that vertices in the center of the polygon are not connected to the remaining unreachable boundary (bottom). (a) The shortest path π_3^r from the right endpoint of the previous window $\omega(a)$ to the right endpoint of b_3 intersects itself. (b) The shortest path π_2^r from the right endpoint of $\omega(a)$ to the right endpoint of b_2 contains a left bend after passing an unreachable vertex that is not connected to the unreachable boundary.

Computing Minimum-Link Paths. Given a sequence $t_1, \ldots, t_k$ of important triangles in a border region B computed as described above, we discuss how a minimum-link path between both sides of e is computed. In particular, we show which modifications to FMLP are necessary to preserve correctness.

Consider the computation of a window from an arbitrary *initial edge* shared by two triangles in $t_1, \ldots, t_k$ as described in Section 6.4.2. Clearly, the subsequence of triangles that is visited until a window is found does not contain multiple occurrences of the same triangle, since this would imply that a straight visibility line intersects it at least twice. Consequently, the fact that triangles may appear several times in $t_1, \ldots, t_k$ does not affect window computation starting at an edge. A similar argument applies when initializing the computation of a subsequent window. Recall that in this step, the visibility cone from the last window to the next initial edge is computed. All triangles considered in this step are intersected by the last window in a certain order (see Section 6.4.2) and because windows are straight lines, we cannot encounter the same triangle twice.

However, the computation of the next window after the initialization step requires some modification, since triangles visited during initialization may reoccur when computing the window from the next initial edge. As a result, the subpaths computed during initialization and when starting from this edge may cross each other; see Figure 6.22a. In this example, the subpath of the right shortest path π_3^r starting at the initial edge b_0 intersects the segment from the right endpoint of the previous window $\omega(a)$ to the right endpoint of b_0. Self-intersections would not pose a problem per se if we generalized the definition of shortest paths to polygons with self-intersections. However, without modifications, FMLP may produce wrong results in certain special cases. Figure 6.22b shows such an example. Although the shortest path from the right endpoint of $\omega(a)$ to the right endpoint of b_2 does not contain crossing edges in this case, its second segment lies in the half plane to the left of the first segment. Hence, the algorithm shown in Figure 6.12 in Section 6.4.2 will falsely remove the last segment. The resulting incorrect path consists of the single segment from the right endpoint of $\omega(a)$ to $r(b_0)$. Clearly, this leads to the construction of an incorrect visibility cone. We say that the last segments of the right shortest paths shown in Figure 6.22 are *visibility-intersecting*, as they reach into the area that is visible from $\omega(a)$. Formally, a segment is visibility-intersecting if it intersects the interior of the hourglass H_0 bounded by the previous window $a' := \omega(a)$, the next initial edge b_0, and the initial left and right shortest path π_0^ℓ and π_0^r; see the shaded area in Figure 6.22a. Visibility-intersecting segments can only occur in the shortest path that corresponds to the unreachable boundary, since the reachable boundary consists of a single component.

In what follows, we show how we can avoid visibility-intersecting segments that may spoil the FMLP algorithm. As argued above, visibility-intersecting segments only occur if a triangle visited during the initialization phase is visited again when computing the next window from a border edge. A conceptually easy way to resolve this issue is to retriangulate parts of the border region, namely, the part called P_0' in Section 6.4.2. Instead, we present an approach that avoids retriangulation by making use of a few simple checks instead (in a sense, it simulates the situation after such a retriangulation). We first show how we can easily detect visibility-intersecting segments. Afterwards, we show that we can simply omit such segments from the corresponding shortest path. As a result, our adaptation is very easy to implement and produces negligible overhead in practice.

Lemma 6.7 claims that for $i \in \{1, \dots, k\}$, a segment that is appended to the shortest path π_i^r is visibility-intersecting if and only if it intersects the previous window a' and is not an endpoint a'. As before, we assume general position. Thus, the window a' and the path π_i^r share no common segment. In practice, such a segment is easily detected and removed from both the path and the window during initialization of π_0^r.

Lemma 6.7. *Given the previous window a', let b_0 be the next initial edge in B. Let t_i, with $i \in \{1, \dots, k\}$, be an important triangle such that the shortest path π_{i-1}^r contains no visibility-intersecting segments and the edge b_i shared by t_i and t_{i+1} is (partially) visible from a' (we set $b_i := b$ in the case $i = k$). The next segment s appended to π_{i-1}^r is visibility-intersecting if and only if s intersects the open line segment a'.*

Proof. Note that we consider the previous window a' to be an open line segment, since its right endpoint coincides with an endpoint of the first segment of π_{i-1}^r, which is clearly not visibility-intersecting.

First, assume the segment $s = pq$ is visibility-intersecting and assume for the sake of contradiction that s does not intersect a'. Since s is visibility-intersecting, it intersects the interior of the hourglass H_0 enclosed by a', b_0, π_0^r, and π_0^ℓ. Since the interior of H_0 contains no vertices (in particular, neither p nor q), the edge s of t_i must intersect the boundary of H_0 at least twice. However, s does not intersect the interior of the edge b_0, since both s and b_0 are edges of triangles. As s does not intersect a' by assumption, it intersects π_0^r or π_0^ℓ. Moreover, since both paths are concave in H_0, s must intersect both paths. (If it intersects any path twice at two points p' and q', the subsegment that connects p' and q' does not intersect the interior of H_0, so s has at least one additional intersection with the boundary of H_0.) But s does not intersect π_0^ℓ, because this would imply that the paths π_i^r and π_0^ℓ have non-empty intersection, contradicting the fact that b_i is visible from a'.

Second, assume that s intersects the open segment a'. Since a' contains no endpoint of s, we know that s intersects the interior of H_0. Hence, the segment s must be visibility-intersecting. □

Next, we show that visibility-intersecting segments can safely be omitted from the shortest path computed by FMLP. Let $t_i, \ldots, t_j$ with $1 \leq i < j \leq k$ be a subsequence of important triangles, such that the edge of t_i appended to π_{i-1}^r by the algorithm is visibility-intersecting, and t_j is the first triangle (i. e., with lowest index $j > i$) such that the edge b_j shared by t_j and t_{j+1} does not intersect the open segment a'; see the sequence of shaded triangles in Figure 6.22b. Thus, all edges $b_i, \ldots, b_{j-1}$ intersect a'. Moreover, π_{i-1}^r and b_j lie on the same side of a' (otherwise, the window a' would cross the reachable boundary of B). We distinguish two cases, depending on whether b_j is visible from a'. We show that in both cases, we can skip the right endpoints of all edges $b_i, \ldots, b_{j-1}$ when updating the path π_{i-1}^r to obtain the correct window.

First, assume that b_j is (partially) visible from a'. We claim that no right endpoint of an edge $b_i, \ldots, b_{j-1}$ is contained in the shortest path π_j^r. To see this, let u denote the last vertex of π_{i-1}^r and w the right endpoint of b_j. Clearly, a' separates u and w from all right endpoints of $b_i, \ldots, b_{j-1}$. Since b_j is visible, this implies that the segment uw crosses all edges $b_i, \ldots, b_{j-1}$. Therefore, it does not intersect the boundary of B and there can be no shorter path from u to w.

Second, assume that b_j is not visible from a'. We claim that no right endpoint v of an edge $b_i, \ldots, b_{j-1}$ is contained in the visibility cone of a'. Assume for contradiction that such an endpoint v is visible from a'. We know that v and the edge b_{i-1} lie on opposite sides of a'. Since v is visible from a', there exists a straight line that crosses a', b_{i-1} and v in this order. Consequently, it must cross a' twice, contradicting the fact that it is a straight line. Since v is not part of the visibility cone from a', it is not relevant for the computation of the next window $\omega(a')$.

In both cases, we can safely ignore right endpoints of all edges $b_i, \ldots, b_{j-1}$. We adapt our algorithm as follows. Before adding a segment to the shortest path that corresponds to the unreachable boundary, we check whether it intersects the previous window a'. If this is the case, we do not add it to the path.

Finally, applying these modifications, we have to clear one last issue to enable correct initialization of the computation of the next window in FMLP. Recall that during this

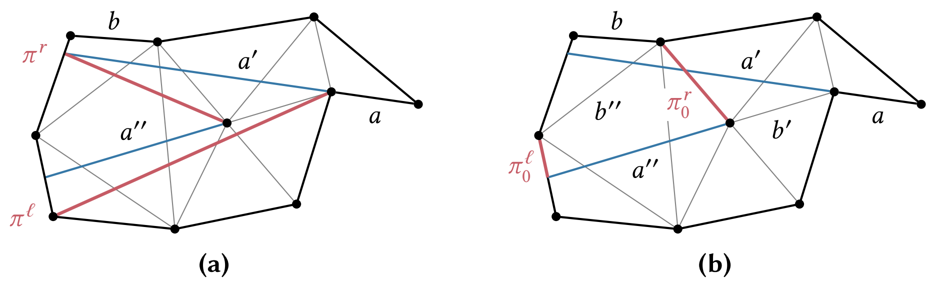


Figure 6.23: Computing initial shortest paths. Assume we want to compute a minimum-link path from a to b, such that the (unreachable) vertex in the center is to the right of this path. (a) The window $a'' = \omega(a')$ computed after the first window a', with final shortest paths (red). (b) The initial shortest paths π_0^ℓ and π_0^r connecting the endpoints of a'' and the next initial edge b'' when computing the next window $\omega(a'')$. To obtain π_0^r, we compute another shortest path from the right endpoint of b', which equals π_0^r when a'' is found.

initialization, one shortest path becomes the suffix of a previous path; see Section 6.4.2. Figure 6.23 illustrates a case where this suffix is not available in the modified algorithm. In this example, we are interested in a minimum-link path between the indicated edges a and b. Figure 6.23a shows the first window $a' = \omega(a)$. Starting from a', the shortest paths π^r and π^ℓ are computed to obtain the next window $a'' = \omega(a')$ (note that the right endpoint of a'' is an unreachable component consisting of a single vertex). To compute the next window $\omega(a'')$ from $\omega(a')$, we first have to compute the initial paths π_0^ℓ and π_0^r, as shown in Figure 6.23b. However, π_0^r consists of a segment that is not present in the previous right shortest path, because it is visibility-intersecting for this path; see the path π^r in Figure 6.23a.

To resolve this problem, we maintain another shortest path π' that starts at the unreachable endpoint of the previous initial edge b', i. e., segments are added as in original FMLP. As argued before, this path does not contain self-intersections. Then, the initial shortest path is a suffix of π', since the only segments omitted from π' are on the shortest path from the previous window $a' = \omega(a)$ to the previous initial edge b'. Clearly, these segments cannot be part of the next initial shortest path, since b' is fully visible from a'. Hence, the first endpoint of the window $\omega(a')$ must be a point in π'.

Remarks. In summary, our algorithm consists of two major steps. The first step traverses the reachable boundary in the triangulation of B. The second step runs a modified version of FMLP, keeping track of the next border edge index to compute the sequence of important triangles on-the-fly. Both steps are modifications of previous algorithms, which maintain their linear running time. While the resulting polygon P may intersect itself, it has at most OPT + 2 segments. To obtain a range polygon P' without self-intersections, we can split P into several non-crossing polygons at intersections. From the resulting smaller polygons, we discard those that contain no vertices of G_p or are fully contained in another polygon. To ensure that P' consists of a single component, according to our primary optimization criterion in Section 6.1, we can also reuse (partial) segments of P to connect the non-crossing components of P'. The number of additional segments is linear in the number of self-intersections, which is small in practice (see Section 6.6.2).

6.6 Experiments

Our experimental study is divided into two main parts. First, we evaluate techniques that compute the reachable subgraph for a source vertex and a range (Section 6.6.1). Second, we consider the computation of the actual range polygon based on the reachable and unreachable part of the network (Section 6.6.2). We also present and analyze typical outputs of our algorithms on real-world road networks (Section 6.6.3).

6.6.1 Computing the Reachable Subgraph

In line with the algorithm descriptions in Section 6.2, we focus on the computation of isochrones. Consequently, we use the instance Eur-DIMACS in most experiments, which provides travel times for all edges in the graph. Nevertheless, we also present results for the EV scenario (using the PHEM model of a Peugeot iOn). Unless mentioned otherwise, all experiments discussed in this section were conducted on machine-p. We always show parallel customization times, but we provide both sequential and parallel query times. Parallel execution uses all available cores. As usual, customization times exclude partitioning, since it is metric-independent. For queries, reported figures are averages of 1 000 queries (per individual range $r \in \mathbb{R}_{\geq 0}$), with source vertices picked uniformly at random. For more details about the experimental setup, see Section 3.4.

Tuning Parameters. We generated multilevel partitions for isoCRP and isoGRASP with PUNCH [Del+11b], whereas we used Buffoon [SS12] to find single-level partitions for isoPHAST. Buffoon works similar to PUNCH, but takes the maximum number $k \in \mathbb{N}$ of cells in the desired partition as input, rather than the maximum cell size; c. f. Section 4.5.3. If necessary, edge partitions were computed from the resulting (vertex) partitions with the approach of Pothen et al. [PSL90, Sch13]. We conducted preliminary studies to obtain reasonable parameters for partitions and search graph compression. For isoCRP and isoGRASP, we use the 4-level partition of Delling et al. [Del+11a], obtained from PUNCH with respective maximum cell sizes of 2^8, 2^{12}, 2^{16}, and 2^{20}. Although Efentakis and Pfoser [EP14] use 16 levels, resorting to a 4-level partition had only minor effects in preliminary experiments (similar observations are made by Efentakis et al. in a subsequent work [EPV15]). For sequential isoPHAST-CD

Table 6.1: Sequential computation of isochrone border edges (Eur-DIMACS). We report parallel customization time and space consumption (space per additional metric is given in brackets, if it differs). The table also shows the average number of vertex scans (# V. Sc.) and running time of sequential queries, using the ranges $r = 100$ and $r = 500$ (in minutes).

	Custom.		$r = 100$ min		$r = 500$ min	
Algorithm	T. [s]	Space [MiB]	# V. Sc.	T. [ms]	# V. Sc.	T. [ms]
isoDijkstra	—	646	460 103	68.32	7 041 260	1 184.06
isoCRP	1.70	900 (138)	100 789	15.44	354 291	60.67
isoGRASP	2.50	1 856 (1 094)	120 327	10.06	387 053	37.77
isoPHAST-CD	26.11	785	440 487	6.09	1 501 455	31.63
isoPHAST-CP	1 221.84	781	626 387	15.02	2 028 703	31.00
isoPHAST-DT	1 079.11	2 935	581 472	9.96	1 813 690	24.80

queries, a partition with $k = 2^{12}$ cells yields best query times. For isoPHAST-CP, we achieve best timings with $k = 2^{11}$ cells. For fewer cells (i. e., coarser partitions), the third query phase scans a large portion of the graph and becomes the bottleneck in both variants. Using more fine-grained partitions, on the other hand, results in a larger core graph, slowing down the second query phase. Consequently, fewer cells ($k = 256$) become favorable for both isoPHAST-CD and isoPHAST-CP when queries are executed in parallel (as only the third phase is parallelized and thus becomes faster). For isoPHAST-DT, we observe similar effects for different values of k. Moreover, search graph compression has a major effect on query times and space consumption. If the number of vertices added to the compressed graph $G_c^\downarrow$ is small, vertices at high levels occur in search graphs of multiple cells, but a large graph $G_c^\downarrow$ causes unnecessary vertex scans. Choosing $k = 2^{14}$ and $|V_c| = 2^{16}$ yields fastest sequential queries, whereas the parameter values $k = 2^{12}$ and $|V_c| = 2^{13}$ provide the fastest parallel queries.

Evaluating Queries. Table 6.1 summarizes the performance of all algorithms discussed in Section 6.2 when computing isochrones on Eur-DIMACS. It shows figures for customization and queries, which are defined by a source vertex together with an indicated range and ask for all corresponding border edges. We report query times for medium ($r = 100$) and long ranges ($r = 500$, this is the hardest range for most approaches, as it results in the largest number of isocontour edges on average). As expected, techniques based on multilevel overlays provide better customization times, while isoPHAST achieves the lowest query times (the best strategy is CD for the medium range and DT for the long range, respectively). Regarding customization, we observe that times provided by isoCRP and isoGRASP are very practical (below three seconds). The lightweight preprocessing of isoPHAST-CD pays off as well, allowing customization in less than 30 seconds. The comparatively high preprocessing times

Table 6.2: Parallel computation of isochrone border edges (Eur-DIMACS). We report figures as in Table 6.1 for the same set of queries, only this time executed in parallel.

	Custom.		r = 100 min		r = 500 min	
Algorithm	T. [s]	Space [MiB]	# V. Sc.	T. [ms]	# V. Sc.	T. [ms]
isoCRP	1.70	900 (138)	100 789	2.73	354 291	7.86
isoGRASP	2.50	1 856 (1 094)	120 327	2.35	387 053	5.93
isoPHAST-CD	38.07	769	917 695	1.61	4 577 630	8.22
isoPHAST-CP	1 432.39	766	943 543	4.47	5 460 207	7.86
isoPHAST-DT	865.50	1 066	913 771	1.74	2 978 899	3.80

of isoPHAST-CP and isoPHAST-DT are mainly due to expensive core contraction. Still, metric-dependent preprocessing is far below half an hour, which is suitable for applications that do not require real-time metric updates. Compared to isoCRP, isoGRASP requires almost an order of magnitude of additional space per metric for the downward graph (which contains about 110 million edges).

Executed sequentially, all speedup techniques take well below 100 ms to answer queries, which is is significantly faster than isoDijkstra. Compared to the multilevel overlay techniques, the number of vertex scans is considerably larger for isoPHAST, yet data access is more cache efficient. As a result, isoPHAST provides faster queries for both limits, with the exception of isoPHAST-CP for small and medium ranges (because the whole core graph is scanned). The performance of isoPHAST-CD is quite notable, providing the fastest queries for medium ranges and decent query times for the higher range. Finally, query times of isoPHAST-DT show best scaling behavior, providing the lowest running times of all approaches for the hardest queries.

Table 6.2 reports parallel times for the same set of random queries. Note that preprocessing times of isoPHAST change due to different parameter choices. Most approaches scale very well with the number of threads, providing a speedup of roughly 8 when using 16 threads. Note that factors (according to Table 6.1 and Table 6.2) are much lower for isoPHAST, since we use tailored partitions for sequential queries. In fact, isoPHAST-DT scales best when run on the same preprocessed data (speedup of 11, not reported in the table), since its sequential workflow (forward CH search, table scan) is very fast. Considering techniques based on multilevel overlays, isoGRASP scales worse than isoCRP (speedup of 6.5 compared to 7.7), probably because it is limited by the memory bandwidth, whereas isoCRP comes with more computational overhead. Consequently, isoGRASP benefits greatly from storing a copy of the downward graph on each NUMA node. As one may expect, speedups are slightly lower for medium-range queries, as there are fewer active cells on average. The isoPHAST approaches yield best query times, which are below 2 ms for medium-range queries and below 4 ms for the long range. To summarize, all algorithms enable queries that are fast

Table 6.3: Computation of border edges with respect to range of an EV (Eur-PTV). As before, customization is run in parallel, while both sequential and parallel queries are reported. Besides the number of threads used in queries (# Th.), we report figures as in Table 6.1.

		Custom.		r = 16 kWh		r = 85 kWh	
Algorithm	# Th.	T. [s]	Space [MiB]	# V. Sc.	T. [ms]	# V. Sc.	T. [ms]
isoDijkstra	1	—	1 558	399 772	63.99	3 931 822	705.28
isoCRP	1	1.72	2 192 (550)	33 602	8.91	114 380	29.56
isoGRASP	1	3.35	4 678 (3 036)	32 620	5.69	107 529	17.30
isoCRP	16	1.72	2 192 (550)	33 602	4.16	114 380	8.70
isoGRASP	16	3.35	4 678 (3 036)	32 620	3.46	107 529	7.55

enough for practical applications, with speedups of more than two orders of magnitude compared to the baseline approach, isoDijkstra.

Finally, we discuss performance of isoCRP and isoGRASP when computing the range of an EV. We focus on these two techniques, because they provide best customization times. As argued in Section 4.4, fast customization is of particular relevance in the context of route planning for EVs. In Table 6.3, we report average sequential and parallel running times of 1 000 random queries on Eur-PTV. Even though the considered graph is largely similar to Eur-DIMACS, we observe significantly faster query times compared to isochrones. This is due to the fact that reachable subgraphs induced by the chosen ranges are smaller (see search spaces reported in the tables). Additionally, we observe that the number of border edges tends to be smaller when computing the range of an EV (even if the reachable subgraph has similar size). A possible explanation for this are differences in the shape of isochrones and isocontours representing the range of an EV: While isochrones reach further on fast routes (e. g., motorways), isocontours representing the range of EVs typically have a more circular shape (motorways allow to move faster, but also consume more energy). Hence, the number of active cells as well as the number of border edges in the output is typically much smaller (by up to a factor of 3) when computing the range of an EV. As a result, speedup decreases when executing our algorithms in parallel (because there are fewer active cells to be processed). Furthermore, we observe that customization effort and space consumption increase. This is only partially explained by the fact that we are dealing with a larger instance. More importantly, two metrics need to be stored in contrast to a single one in the isochrone scenario. Shortcuts also have to store SoC profiles represented by three integers; see Section 4.1.3. Still, customization is very practical, integrating new cost functions within less than four seconds.

Evaluating Scalability. In Figure 6.24, we examine how sequential query times scale with the range. The plot shows running times of random queries asking for

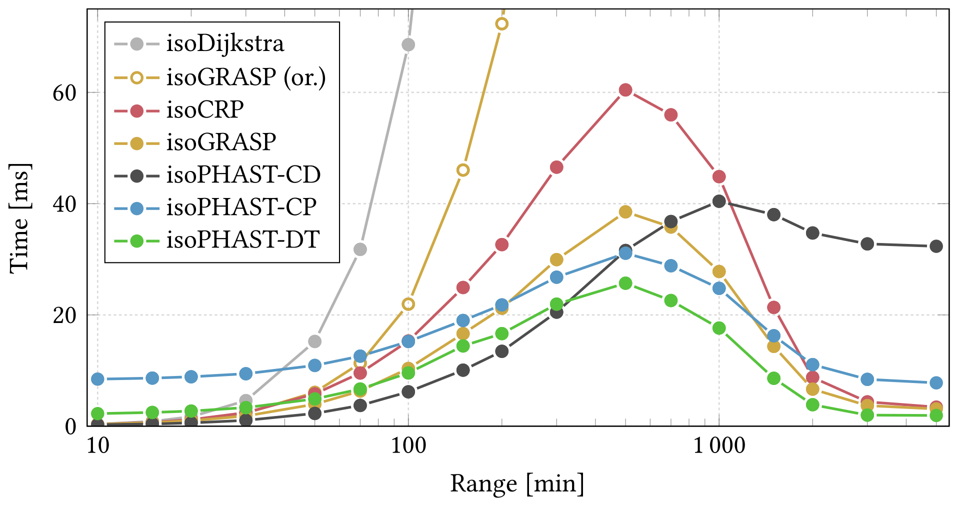


Figure 6.24: Sequential query times of our techniques for various ranges between 10 and 5 000 minutes (which roughly equals the diameter of our input graph, Eur-DIMACS). Each sample corresponds to the average running time of 1 000 queries for the respective range.

isochrones on Eur-DIMACS with indicated ranges. For comparability, we also report sequential query times of original isoGRASP as introduced by Efentakis and Pfoser [EP14], which computes distances to all reachable vertices, but no border edges. Running times of all algorithms except isoDijkstra and original isoGRASP follow a characteristic curve: Timings first increase with the range $r \in \mathbb{R}_{\geq 0}$ (the isochrone frontier is extended, intersecting more active cells), before dropping again once r exceeds 500 minutes (the isochrone reaches the boundary of the network, so the number of active cells decreases). For ranges that exceed 4 800 minutes, all vertices are reachable. Thus, queries become very fast, as there are no active cells. For small values of r, the techniques using multilevel overlays and isoPHAST-CD are the fastest. For these ranges, isoPHAST-CP is slowed down by the linear scan over the core graph (taking about 6 ms, independent of r), while isoPHAST-DT suffers from distance bounds that are not tight. However, since isoDijkstra (run on the core graph) quickly becomes a bottleneck if the range increases, isoPHAST-CD is the slowest of our novel approaches for large values of r, whereas the other strategies based on isoPHAST benefit from good scaling behavior. Considering approaches that use multilevel overlays, our new variant of isoGRASP is up to almost twice as fast as isoCRP, providing a decent trade-off between customization effort and query times. Note that, although isoDijkstra is fast enough for some realistic ranges (below 100 minutes), it is not robust to user inputs on large instances.

Figure 6.25 shows query times subject to different ranges when our query algorithms are executed in parallel. Running times generally follow the same characteristic curve

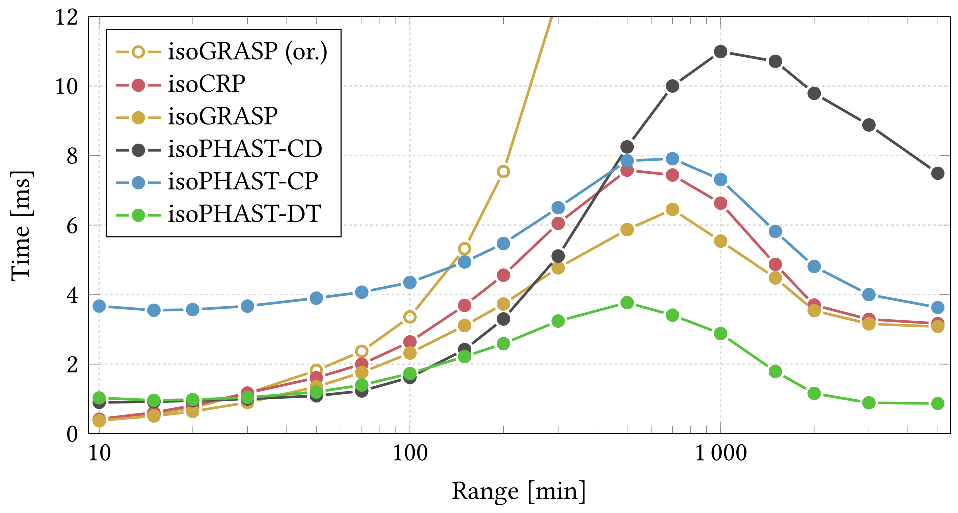


Figure 6.25: Parallel query times of our techniques for various ranges, each given as the average running time for the same set of queries as in Figure 6.24.

as in the sequential scenario. We observe that the linear scan in the second phase of isoPHAST-CP becomes slightly faster (below 4 ms), because the core graph is smaller (due to a different underlying partition). Also, the performance gap between isoCRP and isoGRASP is slightly smaller when using multiple threads. Again, isoPHAST-CD is the best technique for medium-range queries. However, as before, query performance of isoPHAST-CD gets worse if the range increases and isoPHAST-DT becomes the fastest approach for ranges beyond 100 minutes.

Alternative Outputs. Table 6.4 compares query times when computing different outputs, namely, a list of all border edges or a list of all reachable vertices. For medium ranges ($r = 100$ minutes), both sequential and parallel query times increase by less than 10 % when computing the set of reachable vertices instead of border edges. For long ranges ($r = 500$ minutes), where roughly half of the vertices are reachable, sequential and parallel queries slow down by a factor of about 1.5 when computing vertex sets, but they are still significantly faster than the original isoGRASP algorithm. Only when considering the graph diameter as range ($r = 5\,000$ minutes), sequential query times for computing all reachable vertices are significantly slower, since the variants reporting only border edges already terminate after the (very fast) upward phase.

Comparison with Related Work. Since we are not aware of any work solving our compact problem formulation (computing only border edges or reachable vertices), we cannot compare our algorithms directly to competitors. Hence, to validate the

Table 6.4: Impact of different output formats on performance of isoCRP, isoGRASP, and isoPHAST using the CP strategy (Eur-DIMACS). We report average sequential (Seq.) and parallel (Par.) times for 1 000 random queries, as well as output size (# Out.) when computing sets containing all border edges and reachable vertices, respectively.

	Range	Border edges			Reachable vertices		
Algorithm	[min]	# Out.	Seq. [ms]	Par. [ms]	# Out.	Seq. [ms]	Par. [ms]
	100	5 937	15.44	2.73	460 103	15.83	2.77
isoCRP	500	14 718	60.67	7.86	7 041 260	76.35	9.26
	5 000	0	3.42	3.17	18 010 173	46.64	6.64
	100	5 937	10.06	2.35	460 103	11.07	2.50
isoGRASP	500	14 718	37.77	5.93	7 041 260	56.83	7.57
	5 000	0	3.08	3.10	18 010 173	46.09	6.44
	100	5 937	15.02	4.47	460 103	16.40	4.70
isoPHAST	500	14 718	31.00	7.86	7 041 260	49.57	9.67
	5 000	0	7.96	3.61	18 010 173	50.86	7.03

efficiency of our code, we compare our implementations of basic building blocks to the original publications. Table 6.5 reports running times of our implementations of Dijkstra's algorithm, GRASP, PHAST, and RPHAST on one core of machine-s (chosen as it most closely resembles the machines used in the respective original publications [DGW11, EPV15]). For comparison, we report running times (as is) from Delling et al. [DGW11] and Efentakis et al. [EPV15]. One-to-all query times for Dijkstra's algorithm, PHAST, and GRASP are averages of 1 000 random queries on Eur-DIMACS. For the one-to-many scenario (Dijkstra's algorithm and RPHAST), we adopt the methodology of Delling et al. [DGW11]. To determine queries, they pick a center vertex $c \in V$ at random and run Dijkstra's algorithm from c until $|T|$ vertices were scanned, making all scanned vertices the target set T. In a query, distances from a random source $s \in T$ to all vertices in T are requested. In our experiment, we set $|T| = 2^{14}$ (of the scenarios considered by Delling et al. [DGW11], queries of this type most closely resemble the structure of searches in cell-induced subgraphs made by isoPHAST). For RPHAST, we report both target selection and query time.

Even when taking hardware differences into account, we observe that running times of our implementations are similar to the original publications. Note that target selection of RPHAST is even slightly faster.

6.6.2 Computing Range Polygons

We evaluate the algorithms proposed in Sections 6.3–6.5. Given a compact representation of the reachable subgraph, they compute a range polygon that represents

Algorithm	[our]	[orig.]
Dij. (1-to-all)	2 653.18	–
PHAST	144.16	136.92
GRASP	171.11	169.00
Dij. (1-to-many)	7.34	7.43
RPHAST (select)	1.29	1.80
RPHAST (query)	0.16	0.17

Table 6.5: Running times (in milliseconds) of basic building blocks for one-to-all and one-to-many queries (Eur-DIMACS). We compare our implementation of each technique with the respective original publication [DGW11, EPV15].

the actual isocontour. These algorithms do not exploit parallelism, so we conducted our experiments on machine-s. We report experiments on our main benchmark instance, Eur-PTV. To improve spatial locality of the input data, we reorder the vertices of the input graph according to a vertex partition of the graph obtained from PUNCH (using a single level with maximum cell size 2^6).

The planar graph used in our implementation is directed, but stores no cost functions. As mentioned in Section 6.1.1, we add four bounding box vertices in each corner of the embedding, along with eight edges connecting each vertex to the closest vertex of the input graph and the two closest bounding box vertices. During planarization, 293 741 vertices are added and 654 765 edges are split. Note that a dummy vertex may intersect more than two original edges, which explains why the number of split edges is more than twice the number of dummy vertices. They are replaced by 1 591 914 dummy edges (creating 6 294 multi-edges due to overlapping original edges in the given embedding). After planarization, the resulting graph has 22 492 373 vertices and 52 025 261 edges. After triangulating all faces, it has 131 977 245 edges in total.

In what follows, we denote by *RP-RC (Range Polygon from Reachable Extracted Component)* the approach presented in Section 6.5.1, by *RP-TS (Range Polygon from Triangular Separators)* the algorithm from Section 6.5.2, by *RP-CU (Range Polygon from Connected Unreachable Components)* the approach from Section 6.5.3, and by *RP-SI (Range Polygon with Self-Intersections)* the algorithm from Section 6.5.4. We only evaluate Steps 2–4 of the generic method outlined in Section 6.1.2, as the first step (computation of the reachable parts of the graph) was already covered in Section 6.6.1.

Evaluating Queries. We evaluate query scenarios for range visualization of an EV, as well as isochrones. We compare the results provided by the algorithms proposed in Section 6.5. Each algorithm was tested on the same set of 1 000 queries from source vertices picked uniformly at random.

Regarding our primary application, range visualization of EVs, Table 6.6 shows an overview of the results of all heuristics, organized in two blocks. The first considers the medium-range scenario (16 kWh), while the second shows results for long ranges (85 kWh). For RP-SI, the number of components and the complexity are re-

Table 6.6: Running times of range polygon computation for an EV (Eur-PTV). We report, for each algorithm and range, the number of components in the resulting range polygon (Cp.), the complexity of the range polygon (Seg.), the number of self-intersections (Int.), as well as running time of the algorithm. Figures are average values of 1 000 random queries.

	r = 16 kWh				r = 85 kWh			
Algorithm	Cp.	Seg.	Int.	T. [ms]	Cp.	Seg.	Int.	T. [ms]
RP-RC	41	19 396	—	4.50	131	92 554	—	9.46
RP-TS	69	610	—	4.30	219	1 973	—	7.78
RP-CU	41	561	—	10.15	131	1 820	—	25.11
RP-SI	41	549	4.79	7.52	131	1 781	15.06	22.25

ported as is after running the modified FMLP algorithm described in Section 6.5.4 (i. e., resulting polygons have the number of self-intersections reported in the table). Thus, figures slightly change after resolving the intersections (both the number of components and the complexity may increase).

All algorithms perform very well in practice, with timings of 25 ms and below even for large battery capacities. The simpler algorithms, RP-RC and RP-TS are faster by a factor of 2–3 compared to the more sophisticated approaches. On the other hand, we see that range polygons generated by RP-RC have a much higher complexity, exceeding the optimum by more than an order of magnitude. The heuristic RP-TS provides much better results in terms of complexity, but is still outperformed by the other two approaches in this regard. Moreover, the triangular separation increases the number of components by almost a factor of 1.7 (all other approaches in fact compute the minimum number of holes). Regarding the two more involved approaches, RP-CU and RP-SI, we see that the additional effort pays off: Both approaches compute range polygons with the optimal number of components, while keeping the complexity close to the optimum. In fact, we know that each component in the possibly self-intersecting polygon computed by RP-SI requires at most two additional segments compared to an optimal solution (see Section 6.5.4). Taking into account that many small components are triangles (which have optimal complexity), we derive lower bounds on the optimal average complexity of 529 (16 kWh) and 1 720 (85 kWh) for a range polygon with minimum number of components. The average relative error of RP-CU (upper bounded by 6 %) and RP-SI (upper bounded by 4 %) obtained in our experiments is negligible in practice. The number of intersections produced by RP-SI is also rather low, but the majority of computed range polygons contains at least one self-intersection (97.2 % of all range polygons have self-intersections for a range of 85 kWh, not reported in the table).

In Table 6.7, we provide according figures for isochrones, considering a medium range (60 minutes) and the most difficult range from Section 6.6.1 (500 minutes). Again,

Table 6.7: Running times of range polygon computation for isochrones, subject to the indicated ranges. Reported figures are as in Table 6.6. They were obtained by running 1 000 random queries from the same set of source vertices.

	$r = 60\,\text{min}$				$r = 500\,\text{min}$			
Algorithm	Cp.	Seg.	Int.	T. [ms]	Cp.	Seg.	Int.	T. [ms]
RP-RC	53	22 458	—	4.75	231	238 123	—	20.25
RP-TS	151	1 076	—	4.65	694	4 981	—	14.96
RP-CU	53	913	—	12.11	231	4 208	—	65.09
RP-SI	53	881	9.95	8.70	231	4 055	45.80	51.94

these queries are among the hardest in our setting (for longer ranges, the border of the network is reached by many queries). Despite an increase in running time and solution size compared to range visualization for EVs, all approaches still show great performance with average running times of 65 ms and below. As before, the average complexity of range polygons computed by RP-RC is larger compared to other heuristics by about a factor of 50, with range polygons consisting of more than 200 000 segments on average for the long range. This clearly justifies the use of our new algorithms, since a significant decrease of this number is beneficial when efficient rendering or transmission over mobile networks is an issue. Moreover, a smaller number of segments arguably leads to a more appealing visualization for ranges of this order (see Figure 6.28 in Section 6.6.3). For RP-TS, the number of components now exceeds the optimum by about a factor of 3. The approaches RP-CU and RP-SI yield best results, with average relative errors bounded by 7 % and 3 %, respectively.

For the hardest scenario (isochrones, 500 minutes), Table 6.8 reports more detailed information on running times of the different phases of all algorithms. Note that the total running time slightly differs from the sum of all subphases, since it was determined independently. The planarization phase (TP) consists of the linear scans described in Section 6.3.1. Since the same work needs to be done for all approaches, the running time is identical in all cases (bar measurement noise). Of course, the relative effort spent in this step differs per algorithm. For example, it requires more than half of the total running time in case of RP-TS. The time to extract the border regions (BE) applies to all algorithms except RP-TS, where this is done implicitly by checking reachability of vertices while running FMLP. Since RP-RC extracts only the reachable boundary, this phase takes less than half the time compared to RP-CU (the unreachable boundary is typically larger). Finally, RP-SI spends most time in this step, as it runs the extraction on the triangulated graph, which is significantly denser. Recall that RP-SI in fact only extracts the reachable border, similar to RP-RC, so this phase is slower by more than a factor of 2.5. A phase for connecting unreachable components (CC) is only required by RP-CU and takes less time than the extraction of

Table 6.8: Running times of different phases of the algorithms (where applicable) for isochrones (500 minutes). For each algorithm, we report the total running time composed of the running time for transferring the input to the planar graph (TP), extracting the border regions (BE), connecting components (CC), the range polygon computation with minimum-link paths (RP), and the test for self-intersections (SI). Timings are in milliseconds.

Algorithm	TP	BE	CC	RP	SI	Total
RP-RC	8.21	12.01	—	—	—	20.25
RP-TS	8.22	—	—	6.45	—	14.96
RP-CU	8.23	26.66	22.99	7.81	—	65.09
RP-SI	8.20	31.79	—	9.53	2.34	51.94

border regions. Computing the actual range polygon takes a similar amount of time for all approaches that run this phase (6–10 ms). For RP-TS, it is slightly faster, since the algorithm works only on important triangles, reducing the number of visited triangles and simplifying the algorithm. On the other hand, RP-SI is the slowest approach in this phase. This can be explained by the additional overhead caused by modifications described in Section 6.5.4. Moreover, in contrast to RP-TS and RP-CU, there are no artificial edges in the border regions. Hence, windows computed by RP-SI are longer on average, increasing the number of triangles visited by the algorithm.

In summary, we see that extracting the border region takes a major fraction of the total effort for all approaches that construct the border region B explicitly. Despite the algorithmic simplicity of this phase, the size of the border region (more than 500 000 segments on average, not reported in the table) requires a significant amount of work to be done. On the other hand, only a fraction of all triangles in the border regions are actually visited by the FMLP algorithm.

Evaluating Scalability. Figure 6.26 analyzes the scalability of our algorithms, following the methodology of Dijkstra ranks [Bas+16, SS05]. Recall that the Dijkstra rank of a shortest-path query is the number of queue extractions performed by Dijkstra's algorithm, presuming that the algorithm stops once the target is found. Thus, higher ranks reflect harder queries. To obtain queries of different ranks, we executed 1 000 runs of isoDijkstra, with infinite range and from sources chosen uniformly at random. For a search from some source $s \in V$, consider the resource consumption $c(v)$ of the corresponding vertex $v \in V$ that is extracted from the queue in step 2^i of the algorithm. We consider a query from s with range $c(v)$ as a query of rank 2^i (the maximum rank is bounded by the graph size). For each rank in $\{2^1, \ldots, 2^{\lfloor \log n \rfloor}\}$, we evaluate the 1 000 queries generated this way.

Query times of all approaches increase with the Dijkstra rank, which correlates well with the complexity of the border region. Moreover, scaling behavior is similar for all approaches. In accordance with our theoretical findings, our experiment suggests

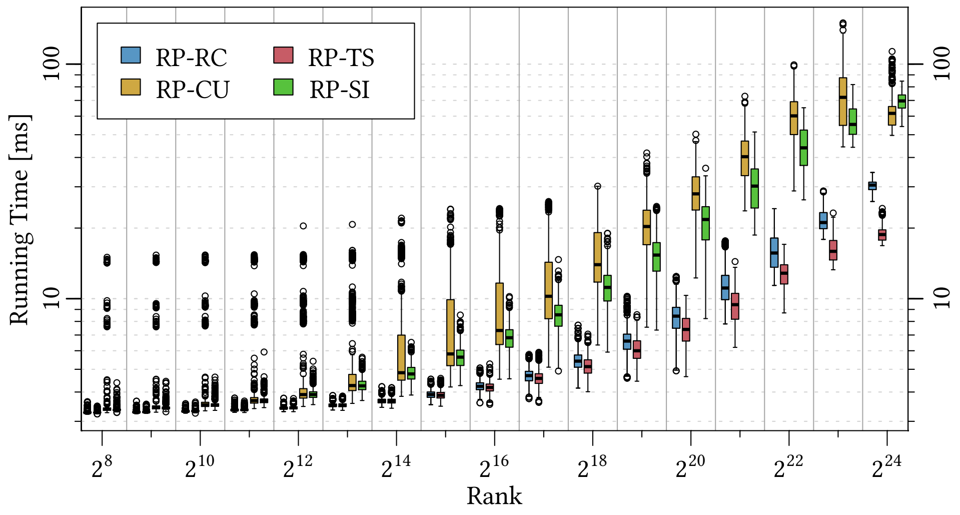


Figure 6.26: Running times of our approaches subject to Dijkstra rank (Eur-PTV). Lower ranks indicate queries of shorter range.

that it increases linearly in the size of the border region for queries beyond a rank of 2^{12}. For queries of lower rank, transferring the input to the planar graph dominates running time, as it is linear in the graph size and thus, independent of the rank. The approach RP-TS is consistently the fastest approach on average for all ranks beyond 2^{16}. Except for a few outliers, our algorithms answer all queries in well below 100 ms. For more local queries (i. e., smaller ranges), query times are much faster (20 ms and below if the rank is at most 2^{20}, corresponding to about a million vertices visited by Dijkstra's algorithm). The more expensive approaches have a higher variance and produce more outliers, which is explained by their more complex phases. For example, the performance of the BFS used in RP-CU heavily depends on how close unreachable components of the border region are in the dual graph.

Evaluating the Computation of Minimum-Link Paths. We take a closer look at the performance of the FMLP algorithm for computing minimum-link paths introduced in Section 6.4. To properly evaluate the algorithm in the context of our experimental setting, we proceed as follows. For a query (defined by source and range), we consider the largest corresponding border region (with respect to the number of segments). To obtain a polygon without holes, as required by the algorithm, we first run our heuristic to connect all unreachable components described in Section 6.5.3. Then, we add an arbitrary border edge to the modified border region and compute a minimum-link path that connects both sides of this edge. Results are shown in Table 6.9. Each scenario is based on the respective sets of random queries used in Table 6.6 and Table 6.7.

Table 6.9: Performance of different minimum-link path algorithms. For each considered scenario, we report the number of segments in the input polygon ($|P|$) and the minimum number of links in the resulting path (Seg.). For Suri's algorithm [Sur86], we show the number of visibility polygon computations (V. Pol.), the total number of segments in the input of these computations (Pol. Seg.), and the total number of visible triangles in these inputs (Trng.). For FMLP, the table provides the number of triangles visited by the algorithm (Trng.) and the running time in milliseconds. Figures are average values of 1 000 queries. Running times exclude the time for triangulating the input polygon, which is part of preprocessing.

			Suri [Sur86]			FMLP	
Range	$\|P\|$	Seg.	V. Pol.	Pol. Seg.	Trng.	Trng.	T. [ms]
16 kWh (EV)	134 049	415	2 010	307 583	48 762	8 901	0.74
60 min (Iso)	135 112	700	3 413	320 244	57 549	11 250	1.05
85 kWh (EV)	357 335	1 328	6 442	850 293	178 574	31 657	3.17
500 min (Iso)	637 224	3 203	15 655	1 547 962	359 969	66 163	6.67

We also compare the performance of FMLP to Suri's algorithm [Sur86], which finds the next window starting from an arbitrary window (or edge) a by computing several visibility polygons as follows. It starts by computing the visibility polygon of a in the polygon bounding all important triangles (as defined in Section 6.4.1) intersected by a. Afterwards, the algorithm iteratively computes new visibility polygons, each time doubling the number of important triangles in the input polygon, until there is an important triangle that is invisible from a. To obtain the actual window, a final visibility polygon is computed for a polygon bounding the same set of important triangles together with all non-important triangles whose closest important triangle (with respect to distance in the dual graph) belongs to this set. Then, the next window is an edge of this visibility polygon.

Clearly, a practical implementation of Suri's algorithm requires a fast subroutine to compute visibility polygons. Moreover, it needs to fill certain degrees of freedom, e. g., generating the input for its subroutine that computes visibility polygons, or determining the window from the resulting visibility polygon. A fair experimental comparison of running times requires a tuned implementation of Suri's algorithm that efficiently fills these degrees of freedom, which is beyond the scope of our work. Instead, Table 6.9 provides measures that are independent of both the machine and the implementation, such as the number of calls to the subroutine and the total number of segments in the generated input polygons. A recent experimental study on visibility polygon computation [Bun+14] proposes a linear-time algorithm for hole-free polygons that is based on a triangulation of the input. It outperforms other approaches in their evaluation because it processes only *visible* triangles. For our purposes, this approach would have to be generalized to compute visibility from windows (rather than just single points). Nevertheless, it is a good candidate for

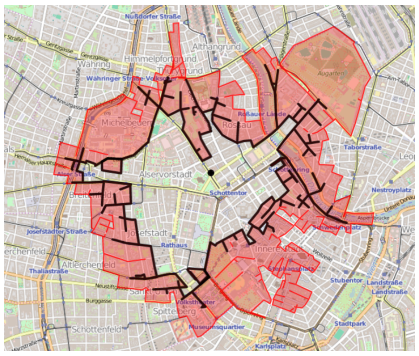
Table 6.10: Results of our algorithms on different instances (Swi-OSM, Ger-OSM), computing isochrones for a range of 60 minutes. We report figures as in Table 6.6. They were obtained by running 1 000 queries with sources picked uniformly at random, as before.

	Switzerland				Germany			
Algorithm	Cp.	Seg.	Int.	T. [ms]	Cp.	Seg.	Int.	T. [ms]
RP-RC	142	258 816	—	15.15	332	223 039	—	16.85
RP-TS	422	2 419	—	8.05	924	5 073	—	13.39
RP-CU	142	1 957	—	65.10	332	4 070	—	68.76
RP-SI	142	1 832	47.05	59.55	332	3 833	95.16	68.53

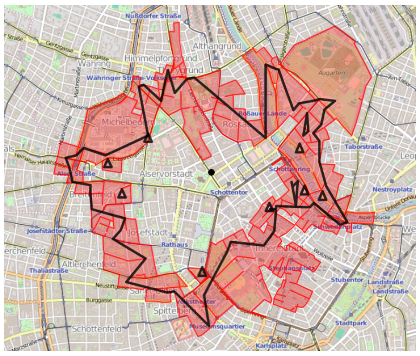
a practical implementation of Suri’s algorithm. Therefore, we also report the total number of *visible* triangles in all polygons constructed by Suri’s algorithm.

For all considered scenarios, Suri’s algorithm requires several thousand calls to the subroutine for visibility polygons. The total number of segments in all polygons computed by Suri’s algorithm is over 1.5 million for the hardest scenario (500 min), which even rules out explicit construction of these polygons for practical applications. In addition to that, the total number of triangles visited by Suri’s algorithm (presuming that it uses the practical visibility polygon algorithm mentioned above) exceeds the number of triangles visited by FMLP by about a factor of 5–6. For FMLP, the workload per visited triangle is very small (updating visibility lines and shortest paths). On the other hand, the visibility polygon algorithm proposed for Suri’s algorithm is recursive [Bun+14] and therefore, possibly less cache efficient. Given that Suri’s algorithm requires additional overhead for constructing input polygons and determining the actual windows from visibility polygons, we conclude that FMLP is much more suitable for practical use.

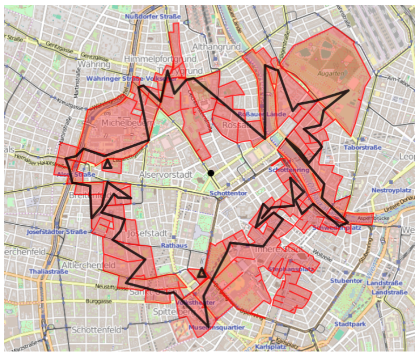
Comparing the different scenarios evaluated in Table 6.9, each represents a certain level of difficulty, with the average complexity of the input polygons ranging from some 100 000 to 600 000 segments. Apparently, the number of visited triangles, the number of segments of the resulting path, and the running time increase with the complexity of the input. However, we also see in Table 6.9 that FMLP performs excellently in practice. Even for input polygons consisting of more than half a million vertices, it computes minimum-link paths in less than 7 ms. Somewhat surprisingly, the isochrone scenario (60 min) is slightly harder to solve for the algorithm than the range scenario (16 kWh), despite a similar input complexity of the input polygons. This can be explained by the different shapes of the respective border regions. Isochrones in road networks reach further on motorways and other fast roads, leading to spike-like shapes in the resulting border regions; see also Figure 6.28 below. Consequently, range polygons for isochrones contain more border edges (c. f. Section 6.6.1), require more segments, and yield the more challenging scenario.



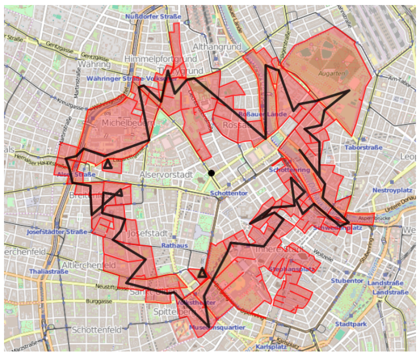
(a)



(b)



(c)



(d)

Figure 6.27: Isochrones in a small-scale example in the city of Vienna, Austria. The black disk in the center is the source vertex and the range is set to a few minutes. Black range polygons represent the isocontours. The corresponding border regions are shaded red. (a) The result of RP-RC. (b) The result of RP-TS. (c) The result of RP-CU. (d) The result of RP-SI.

Other Instances. Finally, we also present experiments for instances extracted from openly available OSM data. We consider the instance Swi-OSM representing the road network of Switzerland, with 3 269 666 vertices and 6 518 469 edges after planarization, and the instance Ger-OSM representing the road network of Germany, with 23 966 527 vertices and 48 398 283 edges after planarization. Recall that these instances have many degree-2 vertices for detailed representation of road curvatures. This explains the relatively large size of the instances (e. g., the graph Ger-OSM contains more vertices than our main benchmark instance Eur-PTV).

Table 6.10 shows results for 1 000 queries on each instance for a medium range of 60 minutes and source vertices picked uniformly at random. (We omit longer ranges, where, particularly for Swi-OSM, major parts of the graph become reachable in every query.) The larger graph sizes are reflected in the average solution sizes and running times. Compared to previous experiments, the decrease in the number of segments of

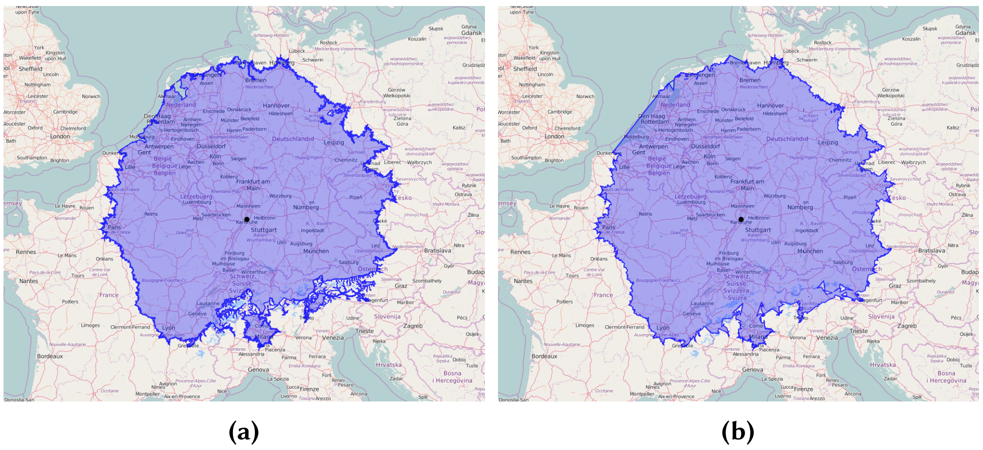


Figure 6.28: Real-world example of a long-range isochrone. It shows the area reachable within five hours from KIT in Karlsruhe, Germany (black disk in the center). (a) The result of RP-RC, with 295 015 segments. (b) The result of RP-CU, with 6 606 segments.

our sophisticated approaches is even more significant. For Swi-OSM, the average result size drops by a factor of more than 100 when using any of the approaches based on minimum-link paths. This is explained by both the large number of degree-2 vertices and the fact that the Swi-OSM instance contains many large faces (representing lakes or mountains). This allows the heuristic to produce long segments; see also Figure 6.28. Lower bounds on the optimal complexity yield similar error bounds as before (at most 7 % on average for our sophisticated techniques). For longer ranges (not reported in the table), running times increase as border region extraction becomes rather expensive due to the graph size. However, even for Ger-OSM, timings were always below 200 ms on average for all tested ranges.

6.6.3 Case Study

To briefly discuss visualization quality, we present outputs of our algorithms for isocontour visualization on Eur-PTV. Figure 6.27 compares results of all four approaches in a small-scale example. The three algorithms based on minimum-link paths produce very similar results, though the resulting polygon contains more holes when using RP-TS. Note that the result of RP-SI contains self-intersections on its left side.

Figure 6.28 shows an isochrone that corresponds to a range of five hours, using the simple RP-RC (Figure 6.28a) and our more sophisticated RP-CU (Figure 6.28b). Comparing the visualization of both approaches, we see that major parts of the isocontours look very similar. However, the number of segments in the range polygon computed by RP-RC and shown in Figure 6.28a is significantly larger, making the isocontour appear more cluttered. At certain points, though, the isocontours differ: The polygon computed by RP-CU contains a long segment at the top left border (covering large parts of the coast of Belgium and the Netherlands), while the isocontour computed by RP-RC stays closer to the shore. This difference is explained by the fact that the sea corresponds to a single huge face in our planar embedding of the input network, which allows the minimum-link path to cover long distances with a single segment. Similar observations can be made at the southern boundary of the reachable area, where many mountains and lakes correspond to large faces in the embedding. Besides long segments, such faces can produce undesirable artifacts, such as spikes. To prevent this, one could add further constraints, e. g., forcing the range polygon to stay reasonably close to the reachable boundary. Such constraints can be implemented by, e. g., slightly shrinking faces (during preprocessing) if their area exceeds a certain threshold. The resulting dummy faces would then be assigned to the unreachable boundary.

6.7 Final Remarks

In this chapter, we proposed several approaches for computing isocontours in large-scale road networks. We identified two major subproblems to achieve this goal: first, the efficient computation of the reachable subgraph and second, the geometric subproblem of computing a polygon that separates the reachable and unreachable parts of the network.

For the first subproblem, we proposed a compact representation of the reachable subgraph based on border edges. We introduced a portfolio of speedup techniques for the resulting problem of computing all border edges. While no single approach turned out to be the best in all considered criteria (preprocessing effort, space consumption, query time, simplicity), the right choice depends on the application. If user-dependent metrics are needed, the fast and lightweight customization of isoCRP is favorable. Fast queries subject to frequent metric updates (e. g., due to real-time traffic) are enabled by our new isoGRASP variant. If customization time below a minute is acceptable and ranges are low, isoPHAST-CD provides even faster query times. The other isoPHAST variants show best scaling behavior, making them suitable for long-range isochrones, or if customizability is not required. Regarding our primary application, range visualization for EVs, we identified isoCRP and isoGRASP as the most suitable candidates, as they provide the best customization times.

Given the subgraph that is reachable from a source within a certain resource limit, the second subproblem boils down to computing a geometric representation of its corresponding border region. We introduced range polygons, following the three major objectives of exact results (reachable and unreachable parts are correctly separated), low complexity (range polygons consist of few segments), and practical performance. We presented three novel algorithms to compute near-optimal solutions in (almost) linear time. Their key ingredient is a new linear-time algorithm for minimum-link paths in simple polygons—the first practical approach to a problem well-studied in theory [Kos+16, MPS14, MRW92, Sur86]. Our experimental evaluation reveals that all approaches compute range polygons within tens of milliseconds on large inputs. Plugging in our speedup techniques for computing the reachable subgraph, our approaches enable isocontour visualization in less than 100 ms in total on our benchmark instance. Thereby, we enable interactive applications even on road networks of continental scale, while metric updates can be integrated within seconds.

Future Work. There are several interesting open issues and room for further improvements. Regarding our proposed speedup techniques, we are interested in integrating the computation of eccentricities into microcode [Del+17, DW13], an optimization technique to accelerate customization of CRP. For isoPHAST, we want to further separate metric-independent preprocessing and metric customization (exploiting, e. g., CCH [DSW16]). We also explore approaches that do not (explicitly) require a partition of the road network. Another direction of research is the speedup of network Voronoi diagram computation [Erw00, Oka+08], where multiple isocontours are grown simultaneously from a set of Voronoi generators. We are also interested in extending our speedup techniques to more involved scenarios, such as multimodal networks.

Considering range polygons, our techniques exploit the fact that the reachable boundary of a border region is always connected, i. e., $|R| = 1$. This might not be the case in related scenarios, such as multi-source isocontours or in multimodal networks, where one can disembark from public transit vehicles only at certain points [Gam+11]. Thus, it would be interesting to know whether our approaches can be extended to the case of $|R| > 1$. Moreover, Gamper et al. [Gam+11] consider the extent to which reachable edges can be passed in their definition of isochrones. This is relevant particularly for short ranges, or if the graph contains very long edges. Hence, we could modify our approaches to handle this slightly different model.

For aesthetic reasons, one could seek to avoid long straight segments or spikes in the range polygon, which are likely to occur in faces encompassing large areas corresponding to, e. g., big lakes or mountains. As discussed in Section 6.6.3, such constraints could be integrated by adding (during preprocessing) artificial boundaries to faces whose area exceeds a certain threshold. On the other hand, one could also aim at further line simplification, at the cost of inexact results. However, such methods should avoid intersections between different components of the range polygon (i. e., maintain its topology) and error measures should consider the resource consumption at parts of the network that are incorrectly classified by the range polygon (vertices that are close in terms of Euclidean distance may be connected by a much longer shortest path in the graph). Reusing ideas from known line simplification algorithms [BKS98, DP73, Fun+17] could be a promising approach. One could also aim at exact approaches based on such line simplification techniques, by incorporating additional constraints that maintain exactness of the resulting range polygon. Finally, another interesting open problem is the consideration of continuous range visualization for a moving vehicle. Instead of computing the isocontours from scratch, one could try to reuse previously computed information.

7 Conclusion

In this thesis, we designed, analyzed, and evaluated novel algorithmic approaches for route planning that explicitly take requirements of EVs into account. Bearing in mind their restricted range and other specific characteristics, such as the ability to recuperate energy while driving, we followed the paradigm of Algorithm Engineering to derive solutions with good performance, both in theory and in practice. We focused on three important aspects in the context of navigation for EVs, which are briefly recapped below in Section 7.1. Afterwards, we highlight interesting directions of future work in Section 7.2.

7.1 Summary

First, we examined *energy-optimal* routes to maximize the range of an EV. We discussed relevant query types and developed different algorithms to answer them. This included *profile queries*, asking for energy-optimal routes for every possible initial SoC, and routes via *charging stations*. On the theoretical side, we showed that these problems can be solved in polynomial time. In particular, profiles mapping SoC at a source vertex to consumption on an energy-optimal path have linear complexity in the input size, which makes efficient profile search possible. In practice, the careful adaptation of *speedup techniques* enabled fast queries, with running times of well below a second or even within milliseconds on our main test instance, depending on the problem setting. Moreover, we introduced a customizable technique that can incorporate global changes in the cost function in only a few seconds.

Second, we dealt with more complex *time-constrained* problems, where both travel time and energy consumption of a route are considered in optimization. We proposed two realistic problem settings, namely, computing shortest feasible routes including intermediate charging stops and shortest feasible routes allowing adaptive speeds. Both settings extend an $\mathcal{NP}$-hard problem and it turned out that even the construction of basic exponential-time algorithms is nontrivial. We suggested speedup techniques based on A* search and CH to improve (empirical) running times of our approaches. Additionally, we presented heuristic variants, which drop correctness for faster queries. Our techniques compute *optimal* results in less than two minutes on our benchmark instance and within seconds for sensible queries, while heuristics retain high-quality solutions at query times below 100 ms.

Third, we considered the problem of quickly and accurately *visualizing* the remaining cruising range of an EV. Also taking account of efficient rendering, we proposed isocontours represented by *range polygons* of low descriptive complexity. We identified two important subproblems: computing the subgraph that is reachable from a given position and computing the range polygon for this subgraph. Regarding the former problem, we introduced a plethora of speedup techniques, providing different tradeoffs in terms of customization overhead, space consumption, and query performance. For the latter, geometric subproblem, we examined a linear-time approximation algorithm for an important special case. Engineering its main component, the computation of a minimum-link path in a simple polygon, we developed a novel algorithm that runs in linear time and is fast in practice. We also used this algorithm as key ingredient of practical heuristics in the general setting. Our evaluation revealed that, altogether, our toolchain enables the computation of long-range isocontours in less than 100 ms, while keeping their descriptive complexity near the optimum.

To summarize, we investigated three important problem settings in the context of route planning for EVs and derived algorithmic methods to solve them. Besides a thorough theoretical analysis of our algorithms, we demonstrated practicality of all approaches in a challenging experimental setting. It turned out that our techniques are able to provide high-quality solutions for all considered problems, with response times that are fast enough even for interactive applications.

Even though algorithmic approaches towards route planning for EVs were our primary motivation, techniques and insights from this thesis may be reused in several related applications. For instance, energy-efficient or consumption-constrained routes are also important for conventional vehicles running on combustion engines, in order to save (monetary) fuel costs [ELB06, KMM11, Neu10]. Finding routes via charging stations to recharge a limited resource is related to problem settings in aircraft routing and crew scheduling [Cor+01, SBW12], as well as route planning for truck drivers [Kle+17]. In such scenarios, maintenance or break periods must be planned in advance to comply with, e. g., government regulations. Finally, our algorithms for range visualization can be readily extended to other applications that require isocontours in transportation networks, e. g., for visualizing the reachable area of a vehicle within a certain time frame.

7.2 Outlook

There are numerous relevant directions of future work that may exploit or build upon the results of this thesis. An obvious next step would be the combination of different presented algorithmic solutions to a single, holistic route planning application. For example, this would require the integration of adaptive speeds and charging stops into a single search [Nik17]. Given that the resulting problem setting is rather complex, one could also further explore heuristic techniques that quickly compute results of near-optimal (empirical) quality.

Another important aspect is the adaptation of realistic models of *turn costs* (in terms of energy consumption), which may result in more sensible algorithm outputs in practice. In particular, taking energy consumption on turns into account can prevent solutions with minor detours that save negligible amounts of energy if turn costs are ignored. Preliminary experiments indicate that established algorithmic techniques for turn costs [Del+17, GV11] can be integrated into our approaches without significant drops in performance. In fact, the solution space may even decrease if minor roads become less attractive due to turn costs.

Further, we assumed in all chapters that travel time and energy consumption are *independent* of the current time of day. In contrast, *time-dependent* route planning [Bat+13, DW09, FHS14] takes historic knowledge about traffic patterns into account and derives edge costs that vary with time to reflect, e. g., peak hours. The integration with time-dependent costs (with respect to both travel time and energy consumption) would render most problems considered in this work much more difficult, but efficient approximation algorithms or heuristics could achieve practical running times in this challenging setting.

One could consider other broader scenarios, such as integrated *multimodal* route planning [Del+13a, DPW15b]. For instance, in a park-and-ride setting, a user might be willing to travel with an EV on the first leg of a journey. Parking and recharging it at a suitable location, the user can continue the journey using other means of (public) transportation. By the choice of a parking lot, a (multimodal) route planning algorithm should then ensure that the return trip is feasible after recharging without long waiting time. Similarly, it would be interesting to extend our methods for fast computation of isocontours to multimodal networks.

Another relevant field considers alternative but closely related vehicle types in route planning, such as *plug-in hybrid electric vehicles* [SMS17, SZ16] or *pedelecs* [Hrn+17, Sto12b]. In contrast to EVs, these vehicles can fall back on an alternative power source (namely, a combustion engine or pedaling) in case the battery nears depletion. Hence, one could investigate problems where battery constraints are softened, such that battery depletion is still avoided (e. g., by penalizing corresponding routes), but resorting to alternative power sources is acceptable if the target cannot be reached otherwise (or only when taking a significant detour).

Finally, we assumed throughout this thesis that SoC is the only state of a battery that affects energy consumption. However, in reality, other factors that depend on the state of the battery, such as its current temperature, influence consumption. Therefore, it would also be worthwhile to consider more complex battery models and propagate their state in search algorithms to model, e. g., a cold start at the beginning of a journey. Although the impact of such a process on overall consumption is presumably small, it would be interesting to compare the results to our (simpler) model.

Bibliography

[AB99] Nina Amenta and Marshall Bern. **Surface Reconstruction by Voronoi Filtering**. *Discrete & Computational Geometry* 22:4, pages 481–504, 1999.
Cited on page 17.

[Abr+11] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. **A Hub-Based Labeling Algorithm for Shortest Paths on Road Networks**. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*. Volume 6630 of Lecture Notes in Computer Science, pages 230–241. Springer, 2011.
Cited on page 10.

[Abr+12a] Ittai Abraham, Daniel Delling, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. **HLDB: Location-Based Services in Databases**. In *Proceedings of the 20th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems (GIS'12)*, pages 339–348. ACM, 2012.
Cited on page 12.

[Abr+12b] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. **Hierarchical Hub Labelings for Shortest Paths**. In *Proceedings of the 20th Annual European Symposium on Algorithms (ESA'12)*. Volume 7501 of Lecture Notes in Computer Science, pages 24–35. Springer, 2012.
Cited on page 10.

[Adl+16] Jonathan D. Adler, Pitu B. Mirchandani, Guoliang Xue, and Minjun Xia. **The Electric Vehicle Shortest-Walk Problem With Battery Exchanges**. *Networks and Spatial Economics* 16:1, pages 155–173, 2016.
Cited on page 16.

[Agr+16] Shubham Agrawal, Hong Zheng, Srinivas Peeta, and Amit Kumar. **Routing Aspects of Electric Vehicle Drivers and their Effects on Network Performance**. *Transportation Research Part D: Transport and Environment* 46, pages 246–266, 2016.
Cited on page 138.

[Ahu+90] Ravindra K. Ahuja, Kurt Mehlhorn, James Orlin, and Robert E. Tarjan. **Faster Algorithms for the Shortest Path Problem**. *Journal of the ACM* 37:2, pages 213–223, 1990.
Cited on page 27.

[AIY13] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. **Fast Exact Shortest-Path Distance Queries on Large Networks by Pruned Landmark Labeling**. In *Proceedings of the 34th ACM SIGMOD International Conference on Management of Data (SIGMOD'13)*, pages 349–360. ACM, 2013.
Cited on page 10.

[Ali+14] Mahnoosh Alizadeh, Hoi-To Wai, Anna Scaglione, Andrea Goldsmith, Yue Yue Fan, and Tara Javidi. **Optimized Path Planning for Electric Vehicle Routing and Charging**. In *Proceedings of the 52nd Annual Allerton Conference on Communication, Control, and Computing (Allerton'14)*, pages 25–32. IEEE, 2014.
Cited on page 16.

[ALS13] Julian Arz, Dennis Luxen, and Peter Sanders. **Transit Node Routing Reconsidered**. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*. Volume 7933 of Lecture Notes in Computer Science, pages 55–66. Springer, 2013.
Cited on page 10.

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. **Network Flows: Theory, Algorithms, and Applications**. Prentice Hall, 1993.
Cited on page 15.

[And15] Simeon Andreev. **Consumption and Travel Time Profiles in Electric Vehicle Routing**. Master's thesis. Karlsruhe Institute of Technology, 2015.
Cited on page 137.

[APV16] Pankaj K. Agarwal, Jiangwei Pan, and Will Victor. **An Efficient Algorithm for Placing Electric Vehicle Charging Stations**. In *Proceedings of the 27th International Symposium on Algorithms and Computation (ISAAC'16)*. Volume 64 of Leibniz International Proceedings in Informatics (LIPIcs), pages 7:1–7:12. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016.
Cited on page 9.

[Art+10a] Andreas Artmeier, Julian Haselmayr, Martin Leucker, and Martin Sachenbacher. **The Optimal Routing Problem in the Context of Battery-Powered Electric Vehicles**. In *Proceedings of the 2nd International Workshop on Constraint Reasoning and Optimization for Computational Sustainability (CROCS'10)*, 2010.
Cited on pages 13, 43, 59, 61.

[Art+10b] Andreas Artmeier, Julian Haselmayr, Martin Leucker, and Martin Sachenbacher. **The Shortest Path Problem Revisited: Optimal Routing for Electric Vehicles**. In *Proceedings of the 33rd Annual German Conference on Advances in Artificial Intelligence (KI'10)*. Volume 6359 of Lecture Notes in Computer Science, pages 309–316. Springer, 2010.
Cited on pages 13, 43, 59, 61.

[Asa+16] Johannes Asamer, Anita Graser, Bernhard Heilmann, and Mario Ruthmair. **Sensitivity Analysis for Energy Demand Estimation of Electric Vehicles**. *Transportation Research Part D: Transport and Environment* 46, pages 182–199, 2016.
Cited on pages 14, 63, 138.

[Ata85] Mikhail J. Atallah. **Some Dynamic Computational Geometry Problems**. *Computers & Mathematics with Applications* 11:12, pages 1171–1181, 1985.
Cited on page 50.

[Bal95] Ivan J. Balaban. **An Optimal Algorithm for Finding Segments Intersections**. In *Proceedings of the 11th Annual Symposium on Computational Geometry (SoCG'95)*, pages 211–219. ACM, 1995.
Cited on page 206.

[Bas+07] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. **Fast Routing in Road Networks with Transit Nodes**. *Science* 316:5824, page 566, 2007.
Cited on page 10.

[Bas+16] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. **Route Planning in Transportation Networks**. In *Algorithm Engineering: Selected Results and Surveys*. Volume 9220 of Lecture Notes in Computer Science, pages 19–80. Springer, 2016.
Cited on pages 1, 9, 41, 87, 173, 179, 243.

[Bat+11] Lucas S. Batista, Felipe Campelo, Frederico G. Guimarães, and Jaime A. Ramírez. **A Comparison of Dominance Criteria in Many-Objective Optimization Problems**. In *Proceedings of the 13th IEEE Congress on Evolutionary Computation (CEC'11)*, pages 2359–2366. IEEE, 2011.
Cited on page 152.

[Bat+13] Gernot V. Batz, Robert Geisberger, Peter Sanders, and Christian Vetter. **Minimum Time-Dependent Travel Times with Contraction Hierarchies**. *ACM Journal of Experimental Algorithmics* 18, pages 1.4:1–1.4:43, 2013.
Cited on pages 11, 13, 29, 49, 86, 177, 255.

[Bau+08] Veronika Bauer, Johann Gamper, Roberto Loperfido, Sylvia Profanter, Stefan Putzer, and Igor Timko. **Computing Isochrones in Multi-Modal, Schedule-Based Transport Networks**. In *Proceedings of the 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'08)*, pages 78:1–78:2. ACM, 2008.
Cited on pages 16, 17, 185.

[Bau+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, pages 2.3:1–2.3:31, 2010.
Cited on page 10.

[Bau+13a] Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. **Energy-Optimal Routes for Electric Vehicles**. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'13)*, pages 54–63. ACM, 2013.
Cited on pages 6, 7, 55.

[Bau+13b] Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. **Energy-Optimal Routes for Electric Vehicles**. Technical report 2013-06. Faculty of Informatics, Karlsruhe Institute of Technology, 2013.
Cited on pages 6, 7.

[Bau+14] Moritz Baum, Julian Dibbelt, Lorenz Hübschle-Schneider, Thomas Pajor, and Dorothea Wagner. **Speed-Consumption Tradeoff for Electric Vehicle Route Planning**. In *Proceedings of the 14th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'14)*. Volume 42 of OpenAccess Series in Informatics (OASIcs), pages 138–151. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2014.
Cited on pages 7, 133, 176, 182.

[Bau+15a] Moritz Baum, Valentin Buchhold, Julian Dibbelt, and Dorothea Wagner. **Fast Computation of Isochrones in Road Networks**. Technical report abs/1512.09090. ArXiv e-prints, 2015.
Cited on pages 6, 7.

[Bau+15b] Moritz Baum, Julian Dibbelt, Andreas Gemsa, Dorothea Wagner, and Tobias Zündorf. **Shortest Feasible Paths with Charging Stops for Battery Electric Vehicles**. In *Proceedings of the 23rd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'15)*, pages 44:1–44:10. ACM, 2015.
Cited on pages 6, 7, 55.

[Bau+16a] Moritz Baum, Thomas Bläsius, Andreas Gemsa, Ignaz Rutter, and Franziska Wegner. **Computing Minimum-Link Separating Polygons in Practice**. In *Proceedings of the 32nd European Workshop on Computational Geometry (EuroCG'16)*, 2016.
Cited on pages 6, 7.

[Bau+16b] Moritz Baum, Thomas Bläsius, Andreas Gemsa, Ignaz Rutter, and Franziska Wegner. **Scalable Exact Visualization of Isocontours in Road Networks via Minimum-Link Paths**. In *Proceedings of the 24th Annual European Symposium on Algorithms (ESA'16).* Volume 57 of Leibniz International Proceedings in Informatics (LIPIcs), pages 44:1–44:17. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016.
Cited on pages 6, 7.

[Bau+16c] Moritz Baum, Thomas Bläsius, Andreas Gemsa, Ignaz Rutter, and Franziska Wegner. **Scalable Isocontour Visualization in Road Networks via Minimum-Link Paths**. Technical report abs/1602.01777. ArXiv e-prints, 2016.
Cited on pages 6, 7.

[Bau+16d] Moritz Baum, Valentin Buchhold, Julian Dibbelt, and Dorothea Wagner. **Fast Exact Computation of Isochrones in Road Networks**. In *Proceedings of the 15th International Symposium on Experimental Algorithms (SEA'16).* Volume 9685 of Lecture Notes in Computer Science, pages 17–32. Springer, 2016.
Cited on pages 6, 7.

[Bau+16e] Moritz Baum, Julian Dibbelt, Andreas Gemsa, and Dorothea Wagner. **Towards Route Planning Algorithms for Electric Vehicles with Realistic Constraints**. *Computer Science – Research and Development* 31:1, pages 105–109, 2016.
Cited on pages 7, 176, 182.

[Bau+16f] Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. **Dynamic Time-Dependent Route Planning in Road Networks with User Preferences**. In *Proceedings of the 15th International Symposium on Experimental Algorithms (SEA'16)*. Volume 9685 of Lecture Notes in Computer Science, pages 33–49. Springer, 2016.
Cited on pages 11, 49, 81, 97, 196.

[Bau+17a] Moritz Baum, Julian Dibbelt, Dorothea Wagner, and Tobias Zündorf. **Modeling and Engineering Constrained Shortest Path Algorithms for Battery Electric Vehicles**. In *Proceedings of the 25th Annual European Symposium on Algorithms (ESA'17)*. Volume 87 of Leibniz International Proceedings in Informatics (LIPIcs), pages 11:1–11:16. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017.
Cited on pages 6, 7.

[Bau+17b] Moritz Baum, Jonas Sauer, Dorothea Wagner, and Tobias Zündorf. **Consumption Profiles in Route Planning for Electric Vehicles: Theory and Applications**. In *Proceedings of the 16th International Symposium on Experimental Algorithms (SEA'17)*. Volume 75 of Leibniz International Proceedings in Informatics (LIPIcs), pages 19:1–19:18. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017.
Cited on pages 6, 7.

[Bau+18] Moritz Baum, Thomas Bläsius, Andreas Gemsa, Ignaz Rutter, and Franziska Wegner. **Scalable Exact Visualization of Isocontours in Road Networks via Minimum-Link Paths**. *Journal of Computational Geometry* 9:1, pages 24–70, 2018.
Cited on pages 6, 7.

[Bau11] Moritz Baum. **On Preprocessing the Arc-Flags Algorithm**. Master's thesis. Karlsruhe Institute of Technology, 2011.
Cited on page 27.

[BD09] Reinhard Bauer and Daniel Delling. **SHARC: Fast and Robust Unidirectional Routing**. *ACM Journal of Experimental Algorithmics* 14, pages 2.4:1–2.4:29, 2009.
Cited on page 10.

[BE05] Karin Brundell-Freij and Eva Ericsson. **Influence of Street Characteristics, Driver Category and Car Performance on Urban Driving Patterns**. *Transportation Research Part D: Transport and Environment* 10:3, pages 213–229, 2005.
Cited on pages 136, 138.

[Bed+16] Luca Bedogni, Luciano Bononi, Marco Di Felice, Alfredo D'Elia, Randolf Mock, Francesco Morandi, Simone Rondelli, Tullio Salmon Cinotti, and Fabio Vergari. **An Integrated Simulation Framework to Model Electric Vehicle Operations and Services**. *IEEE Transactions on Vehicular Technology* 65:8, pages 5900–5917, 2016.
Cited on pages 101, 138.

[Bel58] Richard Bellman. **On a Routing Problem**. *Quarterly of Applied Mathematics* 16:1, pages 87–90, 1958.
Cited on pages 12, 61, 75.

[Ber+08] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. **Computational Geometry: Algorithms and Applications**. Third edition. Springer, 2008.
Cited on pages 206, 208.

[BFM09] Holger Bast, Stefan Funke, and Domagoj Matijevic. **Ultrafast Shortest-Path Queries via Transit Nodes**. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. Volume 74 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 175–192. American Mathematical Society, 2009.
Cited on page 10.

[BKS98] Mark de Berg, Marc van Kreveld, and Stefan Schirra. **Topologically Correct Subdivision Simplification Using the Bandwidth Criterion**. *Cartography and Geographic Information Systems* 25:4, pages 243–257, 1998.
Cited on page 250.

[Bla+16] Marco Blanco, Ralf Borndörfer, Nam-Dung Hoang, Anton Kaier, Adam Schienle, Thomas Schlechte, and Swen Schlobach. **Solving Time Dependent Shortest Path Problems on Airway Networks Using Super-Optimal Wind**. In *Proceedings of the 16th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'16)*. Volume 54 of OpenAccess Series in Informatics (OASIcs), pages 12:1–12:15. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016.
Cited on page 177.

[BM16] Peter Berling and Victor Martínez-de-Albéniz. **Dynamic Speed Optimization in Supply Chains with Stochastic Demand**. *Transportation Science* 50:3, pages 1114–1127, 2016.
Cited on page 15.

[BO79] John L. Bentley and Thomas A. Ottmann. **Algorithms for Reporting and Counting Geometric Intersections**. *IEEE Transactions on Computers* 28:9, pages 643–647, 1979.
Cited on page 206.

[Bra+16] Georg Brandstätter, Claudio Gambella, Markus Leitner, Enrico Malaguti, Filippo Masini, Jakob Puchinger, Mario Ruthmair, and Daniele Vigo. **Overview of Optimization Problems in Electric Car-Sharing System Design and Management**. In *Dynamic Perspectives on Managerial Decision Making*. Volume 22 of Dynamic Modeling and Econometrics in Economics and Finance, pages 441–471. Springer, 2016.
Cited on page 9.

[BS12] Gernot V. Batz and Peter Sanders. **Time-Dependent Route Planning with Generalized Objective Functions**. In *Proceedings of the 20th Annual European Symposium on Algorithms (ESA'12)*. Volume 7501 of Lecture Notes in Computer Science, pages 169–180. Springer, 2012.
Cited on page 11.

[Buc15] Valentin Buchhold. **Fast Computation of Isochrones in Road Networks**. Master's thesis. Karlsruhe Institute of Technology, 2015.
Cited on page 203.

[Bun+14] Francisc Bungiu, Michael Hemmer, John E. Hershberger, Kan Huang, and Alexander Kröller. **Efficient Computation of Visibility Polygons**. In *Proceedings of the 30th European Workshop on Computational Geometry (EuroCG'14)*, 2014.
Cited on pages 245, 246.

[Byk78] Alex Bykat. **Convex Hull of a Finite Set of Points in Two Dimensions**. *Information Processing Letters* 7:6, pages 296–298, 1978.
Cited on page 217.

[CBH11] Peter Conradi, Philipp Bouteiller, and Sascha Hanßen. **Dynamic Cruising Range Prediction for Electric Vehicles**. In *Advanced Microsystems for Automotive Applications 2011: Smart Systems for Electric, Safe and Networked Mobility*, pages 269–277. Springer, 2011.
Cited on pages 16, 17.

[CF14] Domenico Cantone and Simone Faro. **Fast Shortest-Paths Algorithms in the Presence of Few Destinations of Negative-Weight Arcs**. *Journal of Discrete Algorithms* 24, pages 12–25, 2014.
Cited on page 12.

[CGR96] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. **Shortest Paths Algorithms: Theory and Experimental Evaluation**. *Mathematical Programming* 73:2, pages 129–174, 1996.
Cited on page 27.

[CGS99] Boris V. Cherkassky, Andrew V. Goldberg, and Craig Silverstein. **Buckets, Heaps, Lists, and Monotone Priority Queues**. *SIAM Journal on Computing* 28:4, pages 1326–1346, 1999.
Cited on page 27.

[CH66] Kenneth L. Cooke and Eric Halsey. **The Shortest Route Through a Network with Time-Dependent Internodal Transit Times**. *Journal of Mathematical Analysis and Applications* 14:3, pages 493–498, 1966.
Cited on page 11.

[Che+10] Boris V. Cherkassky, Loukas Georgiadis, Andrew V. Goldberg, Robert E. Tarjan, and Renato F. Werneck. **Shortest-Path Feasibility Algorithms: An Experimental Evaluation**. *ACM Journal of Experimental Algorithmics* 14, pages 2.7:1–2.7:37, 2010.
Cited on pages 13, 62.

[CM82] João C. N. Clímaco and Ernesto Q. V. Martins. **A Bicriterion Shortest Path Algorithm**. *European Journal of Operational Research* 11:4, pages 399–404, 1982.
Cited on page 12.

[Coh+03] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. **Reachability and Distance Queries via 2-Hop Labels**. *SIAM Journal on Computing* 32:5, pages 1338–1355, 2003.
Cited on page 10.

[Cor+01] Jean-François Cordeau, Goran Stojković, François Soumis, and Jacques Desrosiers. **Benders Decomposition for Simultaneous Aircraft Routing and Crew Scheduling**. *Transportation Science* 35:4, pages 375–388, 2001.
Cited on page 254.

[Cor+09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. **Introduction to Algorithms**. Third edition. MIT Press, 2009.
Cited on pages 15, 19, 35, 39, 85.

[DAn+12] Gianlorenzo D'Angelo, Mattia D'Emidio, Daniele Frigioni, and Camillo Vitale. **Fully Dynamic Maintenance of Arc-Flags in Road Networks**. In *Proceedings of the 11th International Symposium on Experimental Algorithms (SEA'12)*. Volume 7276 of Lecture Notes in Computer Science, pages 135–147. Springer, 2012.
Cited on page 10.

[Dan63] George B. Dantzig. **Linear Programming and Extensions**. Princeton University Press, 1963.
Cited on pages 9, 82.

[Dea04] Brian C. Dean. **Shortest Paths in FIFO Time-Dependent Networks: Theory and Algorithms**. Technical report. Massachusetts Institute of Technology, 2004.
Cited on pages 11, 13.

[Dea99] Brian C. Dean. **Continuous-Time Dynamic Shortest Path Algorithms**. Master's thesis. Massachusetts Institute of Technology, 1999.
Cited on pages 29, 31.

[Del+09] Daniel Delling, Martin Holzer, Kirill Müller, Frank Schulz, and Dorothea Wagner. **High-Performance Multi-Level Routing**. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. Volume 74 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 73–92. American Mathematical Society, 2009.
Cited on pages 10, 41, 79, 193.

[Del+11a] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. **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.
Cited on page 233.

[Del+11b] Daniel Delling, Andrew V. Goldberg, Ilya Razenshteyn, and Renato F. Werneck. **Graph Partitioning with Natural Cuts**. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS'11)*, pages 1135–1146. IEEE, 2011.
Cited on pages 11, 91, 95, 233.

[Del+13a] Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato F. Werneck. **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.
Cited on pages 11, 255.

[Del+13b] Daniel Delling, Andrew V. Goldberg, Andreas Nowatzyk, and Renato F. Werneck. **PHAST: Hardware-Accelerated Shortest Path Trees**. *Journal of Parallel and Distributed Computing* 73:7, pages 940–952, 2013.
Cited on pages 6, 11, 17, 34, 187, 198.

[Del+14a] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. **Robust Distance Queries on Massive Networks**. In *Proceedings of the 22nd Annual European Symposium on Algorithms (ESA'14)*. Volume 8737 of Lecture Notes in Computer Science, pages 321–333. Springer, 2014.
Cited on page 10.

[Del+14b] Daniel Delling, Andrew V. Goldberg, Ruslan Savchenko, and Renato F. Werneck. **Hub Labels: Theory and Practice**. In *Proceedings of the 13th International Symposium on Experimental Algorithms (SEA'14)*. Volume 8504 of Lecture Notes in Computer Science, pages 259–270. Springer, 2014.
Cited on page 10.

[Del+15] Daniel Delling, Julian Dibbelt, Thomas Pajor, and Renato F. Werneck. **Public Transit Labeling**. In *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*. Volume 9125 of Lecture Notes in Computer Science, pages 273–285. Springer, 2015.
Cited on page 11.

[Del+17] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. **Customizable Route Planning in Road Networks**. *Transportation Science* 51:2, pages 566–591, 2017.
Cited on pages 3, 5, 10, 11, 17, 33, 34, 41, 42, 79, 81, 82, 92, 97, 99, 187, 193, 196, 250, 255.

[Del09] Daniel Delling. **Engineering and Augmenting Route Planning Algorithms**. PhD thesis. Karlsruhe Institute of Technology, 2009.
Cited on page 39.

[Del11] Daniel Delling. **Time-Dependent SHARC-Routing**. *Algorithmica* 60:1, pages 60–94, 2011.
Cited on pages 11, 86.

[Des+16] Guy Desaulniers, Fausto Errico, Stefan Irnich, and Michael Schneider. **Exact Algorithms for Electric Vehicle-Routing Problems with Time Windows**. *Operations Research* 64:6, pages 1388–1405, 2016.
Cited on page 16.

[DGJ09] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson (editors). **The Shortest Path Problem: Ninth DIMACS Implementation Challenge**. Volume 74 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society, 2009.
Cited on page 38.

[DGW11] Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. **Faster Batched Shortest Paths in Road Networks**. 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 52–63. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2011.
Cited on pages 12, 35, 187, 198, 239, 240.

[DI10] Yefim Dinitz and Rotem Itzhak. **Hybrid Bellman-Ford-Dijkstra Algorithm**. Technical report CS-10-04. Ben-Gurion University of the Negev, 2010.
Cited on page 12.

[Dib+13] Julian Dibbelt, Thomas Pajor, Ben Strasser, and Dorothea Wagner. **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.
Cited on page 12.

[Dij59] Edsger W. Dijkstra. **A Note on Two Problems in Connexion with Graphs**. *Numerische Mathematik* 1:1, pages 269–271, 1959.
Cited on pages 9, 26, 59, 118, 192.

[DMS08] Yann Disser, Matthias Müller-Hannemann, and Mathias Schnee. **Multi-Criteria Shortest Paths in Time-Dependent Train Networks**. In *Proceedings of the 7th International Workshop on Experimental Algorithms (WEA'08)*. Volume 5038 of Lecture Notes in Computer Science, pages 347–361. Springer, 2008.
Cited on pages 29, 118.

[DN12] Daniel Delling and Giacomo Nannicini. **Core Routing on Dynamic Time-Dependent Road Networks**. *Informs Journal on Computing* 24:2, pages 187–201, 2012.
Cited on page 11.

[DP73] David H. Douglas and Thomas K. Peucker. **Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature**. *The Canadian Cartographer* 10:2, pages 112–122, 1973.
Cited on page 250.

[DPW09] Daniel Delling, Thomas Pajor, and Dorothea Wagner. **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, 2009.
Cited on page 11.

[DPW15a] Daniel Delling, Thomas Pajor, and Renato F. Werneck. **Round-Based Public Transit Routing**. *Transportation Science* 49:3, pages 591–604, 2015.
Cited on page 12.

[DPW15b] Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. **User-Constrained Multimodal Route Planning**. *ACM Journal of Experimental Algorithmics* 19, pages 3.2:1–3.2:19, 2015.
Cited on pages 11, 127, 255.

[Dre69] Stuart E. Dreyfus. **An Appraisal of Some Shortest-Path Algorithms**. *Operations Research* 17:3, pages 395–412, 1969.
Cited on pages 9, 11, 13, 59.

[DSW16] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. **Customizable Contraction Hierarchies**. *ACM Journal of Experimental Algorithmics* 21, pages 1.5:1–1.5:49, 2016.
Cited on pages 10, 11, 97, 99, 182, 250.

[Duc+08] Matt Duckham, Lars Kulik, Mike Worboys, and Antony Galton. **Efficient Generation of Simple Polygons for Characterizing the Shape of a Set of Points in the Plane**. *Pattern Recognition* 41:10, pages 3224–3236, 2008.
Cited on page 17.

[DW07] Daniel Delling and Dorothea Wagner. **Landmark-Based Routing in Dynamic Graphs**. In *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*. Volume 4525 of Lecture Notes in Computer Science, pages 52–65. Springer, 2007.
Cited on pages 10, 11.

[DW09] Daniel Delling and Dorothea Wagner. **Time-Dependent Route Planning**. In *Robust and Online Large-Scale Optimization.* Volume 5868 of Lecture Notes in Computer Science, pages 207–230. Springer, 2009.
Cited on pages 11, 13, 29, 30, 49, 58, 82, 86, 255.

[DW13] Daniel Delling and Renato F. Werneck. **Faster Customization of Road Networks**. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13).* Volume 7933 of Lecture Notes in Computer Science, pages 30–42. Springer, 2013.
Cited on pages 99, 250.

[DW15] Daniel Delling and Renato F. Werneck. **Customizable Point-of-Interest Queries in Road Networks**. *IEEE Transactions on Knowledge and Data Engineering* 27:3, pages 686–698, 2015.
Cited on pages 5, 12, 17.

[EEP15] Alexandros Efentakis, Christodoulos Efstathiades, and Dieter Pfoser. **COLD. Revisiting Hub Labels on the Database for Large-Scale Graphs**. In *Proceedings of the 14th International Symposium on Spatial and Temporal Databases (SSTD'15).* Volume 9239 of Lecture Notes in Computer Science, pages 22–39. Springer, 2015.
Cited on page 12.

[EEP16] Christodoulos Efstathiades, Alexandros Efentakis, and Dieter Pfoser. **Efficient Processing of Relevant Nearest-Neighbor Queries**. *ACM Transactions on Spatial Algorithms and Systems* 2:3, pages 9:1–9:28, 2016.
Cited on page 12.

[Efe+13a] Alexandros Efentakis, Sotiris Brakatsoulas, Nikos Grivas, Giorgos Lamprianidis, Kostas Patroumpas, and Dieter Pfoser. **Towards a Flexible and Scalable Fleet Management Service**. In *Proceedings of the 6th ACM SIGSPATIAL International Workshop on Computational Transportation Science (IWTCS'13)*, pages 79–84. ACM, 2013.
Cited on pages 17, 186.

[Efe+13b] Alexandros Efentakis, Nikos Grivas, George Lamprianidis, Georg Magenschab, and Dieter Pfoser. **Isochrones, Traffic and DEMOgraphics**. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'13)*, pages 548–551. ACM, 2013.
Cited on pages 5, 16, 17, 185, 186.

[EFS11] Jochen Eisner, Stefan Funke, and Sabine Storandt. **Optimal Route Planning for Electric Vehicles in Large Networks**. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI'11)*, pages 1108–1113. AAAI Press, 2011.
Cited on pages 3, 13, 38, 41, 42, 43, 46, 55, 57, 59, 61, 62, 66, 70, 77, 94, 95, 96, 98.

[EGS10] David Eppstein, Michael T. Goodrich, and Darren Strash. **Linear-Time Algorithms for Geometric Graphs with Sublinearly Many Edge Crossings**. *SIAM Journal on Computing* 39:8, pages 3814–3829, 2010.
Cited on page 206.

[EKS14] Stephan Erb, Moritz Kobitzsch, and Peter Sanders. **Parallel Bi-Objective Shortest Paths Using Weight-Balanced B-Trees with Bulk Updates**. In *Proceedings of the 13th International Symposium on Experimental Algorithms (SEA'14)*. Volume 8504 of Lecture Notes in Computer Science, pages 111–122. Springer, 2014.
Cited on page 12.

[EKS83] Herbert Edelsbrunner, David G. Kirkpatrick, and Raimund Seidel. **On the Shape of a Set of Points in the Plane**. *IEEE Transactions on Information Theory* 29:4, pages 551–559, 1983.
Cited on page 17.

[ELB06] Eva Ericsson, Hanna Larsson, and Karin Brundell-Freij. **Optimizing Route Choice for Lowest Fuel Consumption – Potential Effects of a New Driver Support Tool**. *Transportation Research Part C: Emerging Technologies* 14:6, pages 369–383, 2006.
Cited on page 254.

[EP13] Alexandros Efentakis and Dieter Pfoser. **Optimizing Landmark-Based Routing and Preprocessing**. In *Proceedings of the 6th ACM SIGSPATIAL International Workshop on Computational Transportation Science (IWCTS'13)*, pages 25–30. ACM, 2013.
Cited on pages 10, 32.

[EP14] Alexandros Efentakis and Dieter Pfoser. **GRASP. Extending Graph Separators for the Single-Source Shortest-Path Problem**. In *Proceedings of the 22nd Annual European Symposium on Algorithms (ESA'14)*. Volume 8737 of Lecture Notes in Computer Science, pages 358–370. Springer, 2014.
Cited on pages 5, 6, 11, 17, 35, 187, 192, 196, 197, 233, 237.

[EPV15] Alexandros Efentakis, Dieter Pfoser, and Yannis Vassiliou. **SALT. A Unified Framework for All Shortest-Path Query Variants on Road Networks**. In *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*. Volume 9125 of Lecture Notes in Computer Science, pages 298–311. Springer, 2015.
Cited on pages 10, 12, 17, 233, 239, 240.

[Erw00] Martin Erwig. **The Graph Voronoi Diagram with Applications**. *Networks* 36:3, pages 156–163, 2000.
Cited on page 250.

[ETP12] Alexandros Efentakis, Dimitris Theodorakis, and Dieter Pfoser. **Crowdsourcing Computing Resources for Shortest-Path Computation**. In *Proceedings of the 20th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems (GIS'12)*, pages 434–437. ACM, 2012.
Cited on page 36.

[FAR16] Chiara Fiori, Kyoungho Ahn, and Hesham A. Rakha. **Power-Based Electric Vehicle Energy Consumption Model: Model Development and Validation**. *Applied Energy* 168, pages 257–268, 2016.
Cited on pages 63, 101, 138.

[FHS14] Luca Foschini, John Hershberger, and Subhash Suri. **On the Complexity of Time-Dependent Shortest Paths**. *Algorithmica* 68:4, pages 1075–1097, 2014.
Cited on pages 3, 11, 29, 42, 255.

[Flo+15] Carlos Flores, Vicente Milanés, Joshué Pérez, David González, and Fawzi Nashashibi. **Optimal Energy Consumption Algorithm Based on Speed Reference Generation for Urban Electric Vehicles**. In *Proceedings of the 11th IEEE Intelligent Vehicles Symposium (IV'15)*, pages 730–735. IEEE, 2015.
Cited on pages 14, 101, 133.

[Flo64] Robert W. Floyd. **Algorithm 245: Treesort**. *Communications of the ACM* 7:12, page 701, 1964.
Cited on page 27.

[FLS17] Stefan Funke, Sören Laue, and Sabine Storandt. **Personal Routes with High-Dimensional Costs and Dynamic Approximation Guarantees**. In *Proceedings of the 16th International Symposium on Experimental Algorithms (SEA'17)*. Volume 75 of Leibniz International Proceedings in Informatics (LIPIcs), pages 18:1–18:13. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017.
Cited on page 12.

[FNS15] Stefan Funke, André Nusser, and Sabine Storandt. **Placement of Loading Stations for Electric Vehicles: No Detours Necessary!** *Journal of Artificial Intelligence Research* 53, pages 633–658, 2015.
Cited on page 9.

[FNS16] Stefan Funke, Andre Nusser, and Sabine Storandt. **Placement of Loading Stations for Electric Vehicles: Allowing Small Detours**. In *Proceedings of the 26th International Conference on Automated Planning and Scheduling (ICAPS'16)*, pages 131–139. AAAI Press, 2016.
Cited on page 9.

[Fon13] Matthew W. Fontana. **Optimal Routes for Electric Vehicles Facing Uncertainty, Congestion, and Energy Constraints**. PhD thesis. Massachusetts Institute of Technology, 2013.
Cited on pages 4, 14.

[For56] Lester R. Ford. **Network Flow Theory**. Technical report P-923. Rand Corporation, 1956.
Cited on pages 12, 61, 75.

[Fra+12] Thomas Franke, Isabel Neumann, Franziska Bühler, Peter Cocron, and Josef F. Krems. **Experiencing Range in an Electric Vehicle: Understanding Psychological Barriers**. *Applied Psychology* 61:3, pages 368–391, 2012.
Cited on page 1.

[Fra+16] Thomas Franke, Nadine Rauh, Madlen Günther, Maria Trantow, and Josef F. Krems. **Which Factors Can Protect Against Range Stress in Everyday Usage of Battery Electric Vehicles? Toward Enhancing Sustainability of Electric Mobility Systems**. *Human Factors* 58:1, pages 13–26, 2016.
Cited on page 1.

[FS13] Stefan Funke and Sabine Storandt. **Polynomial-Time Construction of Contraction Hierarchies for Multi-Criteria Objectives**. In *Proceedings of the 15th Meeting on Algorithm Engineering & Experiments (ALENEX'13)*, pages 31–54. SIAM, 2013.
Cited on pages 12, 14, 177.

[FSR06] Liping Fu, Dihua Sun, and Laurence R. Rilett. **Heuristic Shortest Path Algorithms for Transportation Applications: State of the Art**. *Computers & Operations Research* 33:11, pages 3324–3343, 2006.
Cited on page 9.

[FT87] Michael L. Fredman and Robert E. Tarjan. **Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms**. *Journal of the ACM* 34:3, pages 596–615, 1987.
Cited on page 27.

[Fun+17] Stefan Funke, Thomas Mendel, Alexander Miller, Sabine Storandt, and Maria Wiebe. **Map Simplification with Topology Constraints: Exactly and in Practice**. In *Proceedings of the 19th Meeting on Algorithm Engineering & Experiments (ALENEX'16)*, pages 185–196. SIAM, 2017.
Cited on page 250.

[FWL12] Fletcher Foti, Paul Waddell, and Dennis Luxen. **A Generalized Computational Framework for Accessibility: From the Pedestrian to the Metropolitan Scale**. In *Proceedings of the 4th TRB Conference on Innovations in Travel Modeling (ITM'12)*. Transportation Research Board, 2012.
Cited on page 12.

[Gam+11] Johann Gamper, Michael Böhlen, Willi Cometti, and Markus Innerebner. **Defining Isochrones in Multimodal Spatial Networks**. In *Proceedings of the 20th ACM International Conference on Information and Knowledge Management (CIKM'11)*, pages 2381–2384. ACM, 2011.
Cited on pages 16, 185, 192, 250.

[GAP15] Anita Graser, Johannes Asamer, and Wolfgang Ponweiser. **The Elevation Factor: Digital Elevation Model Quality and Sampling Impacts on Electric Vehicle Energy Estimation Errors**. In *Proceedings of the 4th International Conference on Models and Technologies for Intelligent Transportation Systems (MT-ITS'15)*, pages 81–86. IEEE, 2015.
Cited on pages 14, 37.

[Gav+04] Cyril Gavoille, David Peleg, Stéphane Pérennes, and Ran Raz. **Distance Labeling in Graphs**. *Journal of Algorithms* 53:1, pages 85–112, 2004.
Cited on page 10.

[GBI12] Johann Gamper, Michael Böhlen, and Markus Innerebner. **Scalable Computation of Isochrones with Network Expiration**. In *Proceedings of the 24th International Conference on Scientific and Statistical Database Management (SSDBM'12)*. Volume 7338 of Lecture Notes in Computer Science, pages 526–543. Springer, 2012.
Cited on pages 5, 16, 17, 185, 186, 192, 220.

[GBL14] Stefan Grubwinkler, Tobias Brunner, and Markus Lienkamp. **Range Prediction for EVs via Crowd-Sourcing**. In *Proceedings of the 10th IEEE International Vehicle Power and Propulsion Conference (VPPC'14)*. IEEE, 2014.
Cited on page 17.

[Gei+10] Robert Geisberger, Dennis Luxen, Peter Sanders, Sabine Neubauer, and Lars Volker. **Fast Detour Computation for Ride Sharing**. In *Proceedings of the 10th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'10)*. Volume 14 of Open-Access Series in Informatics (OASIcs), pages 88–99. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2010.
Cited on page 12.

[Gei+12a] Robert Geisberger, Michael Rice, Peter Sanders, and Vassilis Tsotras. **Route Planning with Flexible Edge Restrictions**. *ACM Journal of Experimental Algorithmics* 17, pages 1.2:1–1.2:20, 2012.
Cited on page 10.

[Gei+12b] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. **Exact Routing in Large Road Networks Using Contraction Hierarchies**. *Transportation Science* 46:3, pages 388–404, 2012.
Cited on pages 3, 5, 10, 32, 33, 77, 78, 79, 94, 127, 128, 157, 169, 177, 198.

[Gei11] Robert Geisberger. **Advanced Route Planning in Transportation Networks**. PhD thesis. Karlsruhe Institute of Technology, 2011.
Cited on page 12.

[GH05] Andrew V. Goldberg and Chris Harrelson. **Computing the Shortest Path: A* Search Meets Graph Theory**. In *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 156–165. SIAM, 2005.
Cited on pages 10, 32, 99, 183.

[GH89] Leonidas J. Guibas and John E. Hershberger. **Optimal Shortest Path Queries in a Simple Polygon**. *Journal of Computer and System Sciences* 39:2, pages 126–152, 1989.
Cited on page 211.

[GJ79] Michael R. Garey and David S. Johnson. **Computers and Intractability: A Guide to the Theory of NP-Completeness**. W. H. Freeman & Co., 1979.
Cited on pages 12, 19, 223.

[GKS10] Robert Geisberger, Moritz Kobitzsch, and Peter Sanders. **Route Planning with Flexible Objective Functions**. In *Proceedings of the 12th Workshop on Algorithm Engineering & Experiments (ALENEX'10)*, pages 124–137. SIAM, 2010.
Cited on pages 12, 14.

[GKW09] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. **Reach for A*: Shortest Path Algorithms with Preprocessing**. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. Volume 74 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 93–139. American Mathematical Society, 2009.
Cited on page 10.

[GM17] Konstantinos N. Genikomsakis and Georgios Mitrentsis. **A Computationally Efficient Simulation Model for Estimating Energy Consumption of Electric Vehicles in the Context of Route Planning Applications**. *Transportation Research Part D: Transport and Environment* 50, pages 98–118, 2017.
Cited on pages 3, 14, 79.

[GO95] Anka Gajentaan and Mark H. Overmars. **On a Class of $O(n^2)$ Problems in Computational Geometry**. *Computational Geometry* 5:3, pages 165–185, 1995.
Cited on page 219.

[GP14] Michael T. Goodrich and Paweł Pszona. **Two-Phase Bicriterion Search for Finding Fast and Efficient Electric Vehicle Routes**. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'14)*, pages 193–202. ACM, 2014.
Cited on pages 3, 4, 15, 133, 176, 182.

[Gra72] Ronald L. Graham. **An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set**. *Information Processing Letters* 1:4, pages 132–133, 1972.
Cited on pages 125, 129, 155, 217.

[GS15] Dominik Goeke and Michael Schneider. **Routing a Mixed Fleet of Electric and Conventional Vehicle**. *European Journal of Operational Research* 245:1, pages 81–99, 2015.
Cited on page 16.

[Gui+87] Leonidas J. Guibas, John E. Hershberger, Daniel Leven, Micha Sharir, and Robert E. Tarjan. **Linear-Time Algorithms for Visibility and Shortest Path Problems Inside Triangulated Simple Polygons**. *Algorithmica* 2:1, pages 209–233, 1987.
Cited on pages 211, 212.

[Gui+93] Leonidas J. Guibas, John E. Hershberger, Joseph S. B. Mitchell, and J. S. Snoeyink. **Approximating Polygons and Subdivisions with Minimum-Link Paths**. *International Journal of Computational Geometry & Applications* 3:4, pages 383–415, 1993.
Cited on pages 191, 219, 221.

[Gut04] Ronald J. Gutman. **Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks**. In *Proceedings of the 6th Workshop on Algorithm Engineering & Experiments (ALENEX'04)*, pages 100–111. SIAM, 2004.
Cited on pages 10, 83.

[GV11] Robert Geisberger and Christian Vetter. **Efficient Routing in Road Networks with Turn Costs**. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*. Volume 6630 of Lecture Notes in Computer Science, pages 100–111. Springer, 2011.
Cited on pages 182, 255.

[GW05] Andrew V. Goldberg and Renato F. Werneck. **Computing Point-to-Point Shortest Paths from External Memory**. In *Proceedings of the 7th Workshop on Algorithm Engineering & Experiments (ALENEX'05)*, pages 26–40. SIAM, 2005.
Cited on pages 10, 32, 99, 183.

[Han80] Pierre Hansen. **Bicriterion Path Problems**. In *Multiple Criteria Decision Making – Theory and Application*. Volume 177 of Lecture Notes in Economics and Mathematical Systems, pages 109–127. Springer, 1980.
Cited on pages 3, 5, 12, 14, 16, 28, 42, 67, 102, 103.

[Har12] Philipp Harms. **Turn Costs in Energy-Optimal Route Planning for Electric Vehicles**. Bachelor's thesis. Universität zu Lübeck, 2012.
Cited on page 98.

[Hau+09] Stefan Hausberger, Martin Rexeis, Michael Zallinger, and Raphael Luz. **Emission Factors from the Model PHEM for the HBEFA Version 3**. Technical report I-20/2009. University of Technology, Graz, 2009.
Cited on page 37.

[HB15] Gerhard Huber and Klaus Bogenberger. **Long-Trip Optimization of Charging Strategies for Battery Electric Vehicles**. *Transportation Research Record: Journal of the Transportation Research Board* 2497, pages 45–53, 2015.
Cited on page 16.

[Hes+12] Andrea Hess, Francesco Malandrino, Moritz B. Reinhardt, Claudio Casetti, Karin A. Hummel, and Jose M. Barceló-Ordinas. **Optimal Deployment of Charging Stations for Electric Vehicular Networks**. In *Proceedings of the 1st Workshop on Urban Networking (UrbaNe'12)*. ACM, 2012.
Cited on page 9.

[HF14] Frederik Hartmann and Stefan Funke. **Energy-Efficient Routing: Taking Speed into Account**. In *Proceedings of the 37th Annual German Conference on Advances in Artificial Intelligence (KI'14)*. Volume 8736 of Lecture Notes in Computer Science, pages 86–97. Springer, 2014.
Cited on pages 4, 15, 101, 133, 134, 138, 176, 182.

[Hil+09] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. **Fast Point-to-Point Shortest Path Computations with Arc-Flags**. In *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*. Volume 74 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 41–72. American Mathematical Society, 2009.
Cited on page 10.

[HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. **A Formal Basis for the Heuristic Determination of Minimum Cost Paths**. *IEEE Transactions on Systems Science and Cybernetics* 4:2, pages 100–107, 1968.
Cited on pages 3, 5, 9, 31, 78, 118, 152.

[Hol+06] Martin Holzer, Frank Schulz, Dorothea Wagner, and Thomas Willhalm. **Combining Speed-Up Techniques for Shortest-Path Computations**. *ACM Journal of Experimental Algorithmics* 10, pages 2.5:1–2.5:18, 2006.
Cited on page 10.

[Hrn+17] Jan Hrnčíř, Pavol Žilecký, Qing Song, and Michal Jakob. **Practical Multicriteria Urban Bicycle Routing**. *IEEE Transactions on Intelligent Transportation Systems* 18:3, pages 493–504, 2017.
Cited on page 255.

[HS16] Michael Hamann and Ben Strasser. **Graph Bisection with Pareto-Optimization**. In *Proceedings of the 18th Meeting on Algorithm Engineering & Experiments (ALENEX'16)*, pages 90–102. SIAM, 2016.
Cited on page 11.

[HSW09] Martin Holzer, Frank Schulz, and Dorothea Wagner. **Engineering Multilevel Overlay Graphs for Shortest-Path Queries**. *ACM Journal of Experimental Algorithmics* 13, pages 2.5:1–2.5:26, 2009.
Cited on pages 10, 41, 79, 99, 193.

[HZ80] Gabriel Y. Handler and Israel Zang. **A Dual Algorithm for the Constrained Shortest Path Problem**. *Networks* 10:4, pages 293–309, 1980.
Cited on pages 4, 12, 103.

[IBG13] Markus Innerebner, Michael Böhlen, and Johann Gamper. **ISOGA: A System for Geographical Reachability Analysis**. In *Proceedings of the 12th International Conference on Web and Wireless Geographical Information Systems (W2GIS'13)*. Volume 7820 of Lecture Notes in Computer Science, pages 180–189. Springer, 2013.
Cited on pages 16, 17, 185.

[II87] Hiroshi Imai and Masao Iri. **An Optimal Algorithm for Approximating a Piecewise Linear Function**. *Journal of Information Processing* 9:3, pages 159–162, 1987.
Cited on page 209.

[Joh73] Donald B. Johnson. **A Note on Dijkstra's Shortest Path Algorithm**. *Journal of the ACM* 20:3, pages 385–388, 1973.
Cited on pages 12, 59, 61.

[Joh75] Donald B. Johnson. **Priority Queues with Update and Finding Minimum Spanning Trees**. *Information Processing Letters* 4:3, pages 53–57, 1975.
Cited on pages 27, 39.

[Joh77] Donald B. Johnson. **Efficient Algorithms for Shortest Paths in Sparse Networks**. *Journal of the ACM* 24:1, pages 1–13, 1977.
Cited on pages 3, 13, 31, 120, 125.

[JP02] Sungwon Jung and Sakti Pramanik. **An Efficient Path Computation Model for Hierarchically Structured Topographical Road Maps**. *IEEE Transactions on Knowledge and Data Engineering* 14:5, pages 1029–1046, 2002.
Cited on pages 3, 10, 41, 79, 80, 193.

[Jur+14] Tomas Jurik, Arben Cela, Redha Hamouche, René Natowicz, Abdellatif Reama, Silviu-Iulian Niculescu, and Jérôme Julien. **Energy Optimal Real-Time Navigation System**. *IEEE Intelligent Transportation Systems Magazine* 6:3, pages 66–79, 2014.
Cited on page 14.

[KH16] Daniel Krajzewicz and Dirk Heinrichs. **UrMo Accessibility Computer – A Tool for Computing Contour Accessibility Measures**. In *Proceedings of the 8th International Conference on Advances in System Simulation (SIMUL'16)*, pages 56–60. IARIA, 2016.
Cited on page 17.

[Kir+11] Dominik Kirchler, Leo Liberti, Thomas Pajor, and Roberto W. Calvo. **UniALT for Regular Language Constraint Shortest Paths on a Multi-Modal Transportation Network**. In *Proceedings of the 11th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'11)*. Volume 20 of OpenAccess Series in Informatics (OASIcs), pages 64–75. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2011.
Cited on page 11.

[Kle+17] Alexander Kleff, Christian Bräuer, Frank Schulz, Valentin Buchhold, Moritz Baum, and Dorothea Wagner. **Time-Dependent Route Planning for Truck Drivers**. In *Proceedings of the 8th International Conference on Computational Logistics (ICCL'17)*. Volume 10572 of Lecture Notes in Computer Science, pages 110–126. Springer, 2017.
Cited on pages 127, 254.

[Klu+13] Sebastian Kluge, Claudia Sánta, Stefan Dangl, Stefan M. Wild, Martin Brokate, Konrad Reif, and Fritz Busch. **On the Computation of the Energy-Optimal Route Dependent on the Traffic Load in Ingolstadt**. *Transportation Research Part C: Emerging Technologies* 36, pages 97–115, 2013.
Cited on page 13.

[Klu11] Sebastian Kluge. **On the Computation of Fuel-Optimal Paths in Time-Dependent Networks**. PhD thesis. Technische Universität München, 2011.
Cited on page 13.

[KMB81] Lawrence T. Kou, George Markowsky, and Leonard C. Berman. **A Fast Algorithm for Steiner Trees**. *Acta Informatica* 15:2, pages 141–145, 1981.
Cited on page 223.

[KMM11] Samir Khuller, Azarakhsh Malekian, and Julián Mestre. **To Fill or Not to Fill: The Gas Station Problem**. *ACM Transactions on Algorithms* 7:3, pages 36:1–36:16, 2011.
Cited on pages 16, 254.

[KMW10] Philip N. Klein, Shay Mozes, and Oren Weimann. **Shortest Paths in Directed Planar Graphs with Negative Lengths: A Linear-Space $O(n \log^2 n)$-time Algorithm**. *ACM Transactions on Algorithms* 6:2, pages 30:1–30:18, 2010.
Cited on page 12.

[Kno+07] Sebastian Knopp, Peter Sanders, Dominik Schultes, Frank Schulz, and Dorothea Wagner. **Computing Many-to-Many Shortest Paths Using Highway Hierarchies**. In *Proceedings of the 9th Workshop on Algorithm Engineering & Experiments (ALENEX'07)*, pages 36–45. SIAM, 2007.
Cited on page 11.

[Kob+11] Yuichi Kobayashi, Noboru Kiyama, Hirokazu Aoshima, and Masamori Kashiyama. **A Route Search Method for Electric Vehicles in Consideration of Range and Locations of Charging Stations**. In *Proceedings of the 7th IEEE Intelligent Vehicles Symposium (IV'11)*, pages 920–925. IEEE, 2011.
Cited on page 15.

[Kon+15] Spyros Kontogiannis, George Michalopoulos, Georgia Papastavrou, Andreas Paraskevopoulos, Dorothea Wagner, and Christos Zaroliagis. **Analysis and Experimental Evaluation of Time-Dependent Distance Oracles**. In *Proceedings of the 17th Meeting on Algorithm Engineering & Experiments (ALENEX'15)*, pages 147–158. SIAM, 2015.
Cited on page 11.

[Kon+16] Spyros Kontogiannis, George Michalopoulos, Georgia Papastavrou, Andreas Paraskevopoulos, Dorothea Wagner, and Christos Zaroliagis. **Engineering Oracles for Time-Dependent Road Networks**. In *Proceedings of the 18th Meeting on Algorithm Engineering & Experiments (ALENEX'16)*, pages 1–14. SIAM, 2016.
Cited on page 11.

[Kos+16] Irina Kostitsyna, Maarten Löffler, Valentin Polishchuk, and Frank Staals. **On the Complexity of Minimum-Link Path Problems**. In *Proceedings of the 32nd Annual Symposium on Computational Geometry (SoCG'16)*. Volume 51 of Leibniz International Proceedings in Informatics (LIPIcs), pages 49:1–49:16. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016.
Cited on pages 219, 250.

[KSG14] Nikolaus Krismer, Günter Specht, and Johann Gamper. **Incremental Calculation of Isochrones Regarding Duration**. In *Proceedings of the 26th GI-Workshop on Foundations of Databases (GvDB'14)*. Volume 1313 of CEUR Workshop Proceedings, pages 41–46. CEUR-WS.org, 2014.
Cited on page 17.

[KZ16] Spyros Kontogiannis and Christos Zaroliagis. **Distance Oracles for Time-Dependent Networks**. *Algorithmica* 74:4, pages 1404–1434, 2016.
Cited on page 11.

[Lau04] Ulrich Lauther. **An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background**. In *Geoinformation und Mobilität – von der Forschung zur praktischen Anwendung*. Volume 22 of IfGI prints, pages 219–230. Institut für Geoinformatik, Universität Münster, 2004.
Cited on page 10.

[LGR07] Shieu Hong Lin, Nate Gertsch, and Jennifer R. Russell. **A Linear-Time Algorithm for Finding Optimal Vehicle Refueling Policies**. *Operations Research Letters* 35:3, pages 290–296, 2007.
Cited on page 16.

[LL12] James Larminie and John Lowry. **Electric Vehicle Technology Explained, 2nd Edition**. John Wiley & Sons, 2012.
Cited on pages 101, 138.

[LLS16] Chung-Shou Liao, Shang-Hung Lu, and Zuo-Jun Max Shen. **The Electric Vehicle Touring Problem**. *Transportation Research Part B: Methodological* 86, pages 163–180, 2016.
Cited on pages 15, 16.

[LSS17] Yaqiong Liu, Hock Soon Seah, and Guochu Shou. **Constrained Energy-Efficient Routing in Time-Aware Road Networks**. *GeoInformatica* 21:1, pages 89–117, 2017.
Cited on page 14.

[Lv+16] Mingsong Lv, Nan Guan, Ye Ma, Dong Ji, Erwin Knippel, Xue Liu, and Wang Yi. **Speed Planning for Solar-Powered Electric Vehicles**. In *Proceedings of the 7th International Conference on Future Energy Systems (e-Energy'16)*, pages 6:1–6:10. ACM, 2016.
Cited on pages 14, 63, 138.

[LWL14] Chensheng Liu, Jing Wu, and Chengnian Long. **Joint Charging and Routing Optimization for Electric Vehicle Navigation Systems**. In *Proceedings of the 19th International Federation of Automatic Control World Congress (IFAC'14)*. Volume 47 of IFAC Proceedings Volumes, pages 9611–9616. Elsevier, 2014.
Cited on page 16.

[Mar84] Ernesto Q. V. Martins. **On a Multicriteria Shortest Path Problem**. *European Journal of Operational Research* 16:2, pages 236–245, 1984.
Cited on pages 12, 14, 28, 67, 102, 103.

[Mas+14] Michail Masikos, Konstantinos Demestichas, Evgenia Adamopoulou, and Michael Theologou. **Machine-Learning Methodology for Energy Efficient Routing**. *IET Intelligent Transport Systems* 8:3, pages 255–265, 2014.
Cited on pages 3, 13, 79.

[MCB14] Joris Maervoet, Patrick De Causmaecker, and Greet Vanden Berghe. **Fast Approximation of Reach Hierarchies in Networks**. In *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'14)*, pages 441–444. ACM Press, 2014.
Cited on page 10.

[Meh88] Kurt Mehlhorn. **A Faster Approximation Algorithm for the Steiner Problem in Graphs**. *Information Processing Letters* 27:3, pages 125–128, 1988.
Cited on page 223.

[MG10] Sarunas Marciuska and Johann Gamper. **Determining Objects within Isochrones in Spatial Network Databases**. In *Proceedings of the 14th East European Conference on Advances in Databases and Information Systems (ADBIS'10)*. Volume 6295 of Lecture Notes in Computer Science, pages 392–405. Springer, 2010.
Cited on pages 17, 186, 219, 220.

[MM12] Enrique Machuca and Lawrence Mandow. **Multiobjective Heuristic Search in Road Maps**. *Expert Systems with Applications* 39:7, pages 6435–6445, 2012.
Cited on page 12.

[Mog15] Khashayar E. Moghadam. **Optimale Routen für E-Fahrzeuge mit vorreservierten E-Ladestationen**. Diploma thesis. Karlsruhe Institute of Technology, 2015.
Cited on page 183.

[Moh+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, pages 2.8:1–2.8:29, 2006.
Cited on page 10.

[Mon+17] Alejandro Montoya, Christelle Guéret, Jorge E. Mendoza, and Juan G. Villegas. **The Electric Vehicle Routing Problem with Nonlinear Charging Function**. *Transportation Research Part B: Methodological* 103, pages 87–110, 2017.
Cited on pages 16, 105, 106.

[Moo59] Edward F. Moore. **The Shortest Path Through a Maze**. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.
Cited on pages 12, 61, 75.

[MP10] Lawrence Mandow and José-Luis Pérez-de-la-Cruz. **Multiobjective A* Search with Consistent Heuristics**. *Journal of the ACM* 57:5, pages 27:1–27:24, 2010.
Cited on pages 12, 118, 152.

[MPS14] Joseph S. B. Mitchell, Valentin Polishchuk, and Mikko Sysikaski. **Minimum-Link Paths Revisited**. *Computational Geometry* 47:6, pages 651–667, 2014.
Cited on pages 219, 250.

[MRW92] Joseph S. B. Mitchell, Günter Rote, and Gerhard Woeginger. **Minimum-Link Paths Among Obstacles in the Plane**. *Algorithmica* 8:1, pages 431–459, 1992.
Cited on pages 219, 250.

[MS08] Kurt Mehlhorn and Peter Sanders. **Algorithms and Data Structures: The Basic Toolbox**. Springer, 2008.
Cited on pages 19, 222.

[MS10] Matthias Müller-Hannemann and Stefan Schirra (editors). **Algorithm Engineering: Bridging the Gap between Algorithm Theory and Practice**. Volume 5971 of Lecture Notes in Computer Science. Springer, 2010.
Cited on page 2.

[MW01] Matthias Müller-Hannemann and Karsten Weihe. **Pareto Shortest Paths is Often Feasible in Practice**. In *Proceedings of the 5th International Workshop on Algorithm Engineering (WAE'01)*. Volume 2141 of Lecture Notes in Computer Science, pages 185–197. Springer, 2001.
Cited on page 12.

[MW06] Matthias Müller-Hannemann and Karsten Weihe. **On the Cardinality of the Pareto Set in Bicriteria Shortest Path Problems**. *Annals of Operations Research* 147:1, pages 269–286, 2006.
Cited on page 12.

[Nan+12] Giacomo Nannicini, Daniel Delling, Leo Liberti, and Dominik Schultes. **Bidirectional A* Search on Time-Dependent Road Networks**. *Networks* 59:2, pages 240–251, 2012.
Cited on page 11.

[Neu10] Sabine Neubauer. **Planung energieeffizienter Routen in Straßennetzwerken**. Diploma thesis. Karlsruhe Institute of Technology, 2010.
Cited on pages 14, 254.

[Nic66] Alastair J. Nicholson. **Finding the Shortest Route between Two Points in a Network**. *The Computer Journal* 9:3, pages 275–280, 1966.
Cited on page 9.

[Nik17] Patrick Niklaus. **A Unified Framework for Electric Vehicle Routing**. Master's thesis. Karlsruhe Institute of Technology, 2017.
Cited on pages 182, 254.

[Oka+08] Atsuyuki Okabe, Toshiaki Satoh, Takehiro Furuta, Atsuo Suzuki, and Kimie Okano. **Generalized Network Voronoi Diagrams: Concepts, Computational Methods, and Applications**. *International Journal of Geographical Information Science* 22:9, pages 965–994, 2008.
Cited on page 250.

[OMS00] David O'Sullivan, Alastair Morrison, and John Shearer. **Using Desktop GIS for the Investigation of Accessibility by Public Transport: An Isochrone Approach**. *International Journal of Geographical Information Science* 14:1, pages 85–104, 2000.
Cited on pages 16, 17, 185.

[OP14a] Peter Ondrúška and Ingmar Posner. **Probabilistic Attainability Maps: Efficiently Predicting Driver-Specific Electric Vehicle Range**. In *Proceedings of the 10th IEEE Intelligent Vehicles Symposium (IV'14)*, pages 1169–1174. IEEE, 2014.
Cited on pages 16, 17.

[OP14b] Peter Ondrúška and Ingmar Posner. **The Route Not Taken: Driver-Centric Estimation of Electric Vehicle Range**. In *Proceedings of the 24th International Conference on Automated Planning and Scheduling (ICAPS'14)*, pages 413–420. AAAI Press, 2014.
Cited on pages 16, 17.

[OR91] Ariel Orda and Raphael Rom. **Minimum Weight Paths in Time-Dependent Networks**. *Networks* 21:3, pages 295–319, 1991.
Cited on pages 29, 31.

[Paj13] Thomas Pajor. **Algorithm Engineering for Realistic Journey Planning in Transportation Networks**. PhD thesis. Karlsruhe Institute of Technology, 2013.
Cited on pages 39, 226.

[PCW16] Sepideh Pourazarm, Christos G. Cassandras, and Tao Wang. **Optimal Routing and Charging of Energy-Limited Vehicles in Traffic Networks**. *International Journal of Robust and Nonlinear Control* 26:6, pages 1325–1350, 2016.
Cited on page 16.

[Pel+17] Samuel Pelletier, Ola Jabali, Gilbert Laporte, and Marco Veneroni. **Battery Degradation and Behaviour for Electric Vehicles: Review and Numerical Analyses of Several Models**. *Transportation Research Part B: Methodological* 103, pages 158–187, 2017.
Cited on pages 105, 106.

[PJL16] Samuel Pelletier, Ola Jabali, and Gilbert Laporte. **Goods Distribution with Electric Vehicles: Review and Research Perspectives**. *Transportation Science* 50:1, pages 3–22, 2016.
Cited on page 9.

[Poh71] Ira Pohl. **Bi-Directional Search**. In *Proceedings of the 6th Annual Machine Intelligence Workshop*, pages 124–140. Edinburgh University Press, 1971.
Cited on pages 10, 31, 32.

[PS08] Patrice Perny and Olivier Spanjaar. **Near Admissible Algorithms for Multiobjective Search**. In *Proceedings of the 18th European Conference on Artificial Intelligence (ECAI'08)*, pages 490–494. IOS Press, 2008.
Cited on page 14.

[PSL90] Alex Pothen, Horst D. Simon, and Kang-Pu Liou. **Partitioning Sparse Matrices with Eigenvectors of Graphs**. *SIAM Journal on Matrix Analysis and Applications* 11:3, pages 430–452, 1990.
Cited on page 233.

[Pyr+08] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. **Efficient Models for Timetable Information in Public Transportation Systems**. *ACM Journal of Experimental Algorithmics* 12, pages 2.4:1–2.4:39, 2008.
Cited on page 11.

[QE16] Jiani Qian and Richard Eglese. **Fuel Emissions Optimization in Vehicle Routing Problems with Time-Varying Speeds**. *European Journal of Operational Research* 248:3, pages 840–848, 2016.
Cited on page 15.

[QZ11] Hua Qin and Wensheng Zhang. **Charging Scheduling with Minimal Waiting in a Network of Electric Vehicles and Charging Stations**. In *Proceedings of the 8th ACM International Workshop on Vehicular Inter-Networking (VANET'11)*, pages 51–60. ACM, 2011.
Cited on page 183.

[RE09] Andrea Raith and Matthias Ehrgott. **A Comparison of Solution Strategies for Biobjective Shortest Path Problems**. *Computers & Operations Research* 36:4, pages 1299–1331, 2009.
Cited on pages 12, 29.

[Sac+11] Martin Sachenbacher, Martin Leucker, Andreas Artmeier, and Julian Haselmayr. **Efficient Energy-Optimal Routing for Electric Vehicles**. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI'11)*, pages 1402–1407. AAAI Press, 2011.
Cited on pages 13, 41, 62, 63, 98.

[San09] Peter Sanders. **Algorithm Engineering – An Attempt at a Definition**. In *Efficient Algorithms*. Volume 5760 of Lecture Notes in Computer Science, pages 321–340. Springer, 2009.
Cited on page 2.

[Sau15] Jonas Sauer. **Energy-Optimal Routes for Electric Vehicles with Charging Stops**. Bachelor's thesis. Karlsruhe Institute of Technology, 2015.
Cited on page 69.

[SBW12] Olivia J. Smith, Natashia Boland, and Hamish Waterer. **Solving Shortest Path Problems with a Weight Constraint and Replenishment Arcs**. *Computers & Operations Research* 39:5, pages 964–984, 2012.
Cited on pages 3, 4, 15, 254.

[SC07] Jan Scheurer and Carey Curtis. **Accessibility Measures: Overview and Practical Applications**. Working paper no. 4. Curtin University of Technology, 2007.
Cited on page 16.

[Sch13] Christian Schulz. **High Quality Graph Partitioning**. PhD thesis. Karlsruhe Institute of Technology, 2013.
Cited on page 233.

[SDK17] Timothy M. Sweda, Irina S. Dolinskaya, and Diego Klabjan. **Adaptive Routing and Recharging Policies for Electric Vehicles**. *Transportation Science* 51:4, pages 1326–1348, 2017.
Cited on pages 16, 106, 183.

[SF12] Sabine Storandt and Stefan Funke. **Cruising with a Battery-Powered Vehicle and Not Getting Stranded**. In *Proceedings of the 26th AAAI Conference on Artificial Intelligence (AAAI'12)*, pages 1628–1634. AAAI Press, 2012.
Cited on pages 15, 41, 67.

[SHS11] Walter J. Sweeting, Allan R. Hutchinson, and Shaun D. Savage. **Factors Affecting Electric Vehicle Energy Consumption**. *International Journal of Sustainable Engineering* 4:3, pages 192–201, 2011.
Cited on pages 41, 138.

[SK12] Timothy M. Sweda and Diego Klabjan. **Finding Minimum-Cost Paths for Electric Vehicles**. In *Proceedings of the 1st International Electric Vehicle Conference (IEVC'12)*. IEEE, 2012.
Cited on page 16.

[Skr00] Anders J. V. Skriver. **A Classification of Bicriterion Shortest Path (BSP) Algorithms**. *Asia-Pacific Journal of Operational Research* 17:2, pages 199–212, 2000.
Cited on pages 12, 29.

[SL15] René Schönfelder and Martin Leucker. **Abstract Routing Models and Abstractions in the Context of Vehicle Routing**. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI'15)*, pages 2639–2645. AAAI Press, 2015.
Cited on pages 13, 29.

[SLW14] René Schönfelder, Martin Leucker, and Sebastian Walther. **Efficient Profile Routing for Electric Vehicles**. In *Proceedings of the 1st International Conference on Internet of Vehicles (IOV'14)*. Volume 8662 of Lecture Notes in Computer Science, pages 21–30. Springer, 2014.
Cited on pages 13, 41, 97.

[SM13] Peter Sanders and Lawrence Mandow. **Parallel Label-Setting Multi-Objective Shortest Path Search**. In *Proceedings of the 27th IEEE International Parallel and Distributed Processing Symposium (IPDPS'13)*, pages 215–224. IEEE, 2013.
Cited on page 12.

[SMS17] Martin Strehler, Sören Merting, and Christian Schwan. **Energy-Efficient Shortest Routes for Electric and Hybrid Vehicles**. *Transportation Research Part B: Methodological* 103, pages 111–135, 2017.
Cited on pages 15, 16, 98, 133, 176, 183, 255.

[Som14] Christian Sommer. **Shortest-Path Queries in Static Networks**. *ACM Computing Surveys* 46:4, pages 45:1–45:31, 2014.
Cited on page 9.

[SRB16] Marcelo de Souza, Marcus Ritt, and Ana L. C. Bazzan. **A Bi-Objective Method of Traffic Assignment for Electric Vehicles**. In *Proceedings of the 19th International Conference on Intelligent Transportation Systems (ITSC'16)*, pages 2319–2324. IEEE, 2016.
Cited on page 15.

[SS05] Peter Sanders and Dominik Schultes. **Highway Hierarchies Hasten Exact Shortest Path Queries**. In *Proceedings of the 13th Annual European Conference on Algorithms (ESA'05)*. Volume 3669 of Lecture Notes in Computer Science, pages 568–579. Springer, 2005.
Cited on pages 87, 173, 179, 243.

[SS12] Peter Sanders and Christian Schulz. **Distributed Evolutionary Graph Partitioning**. In *Proceedings of the 14th Meeting on Algorithm Engineering & Experiments (ALENEX'12)*, pages 16–29. SIAM, 2012.
Cited on pages 11, 233.

[SS15] Aaron Schild and Christian Sommer. **On Balanced Separators in Road Networks**. In *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA'15)*. Volume 9125 of Lecture Notes in Computer Science, pages 286–297. Springer, 2015.
Cited on page 11.

[SSG14] Michael Schneider, Andreas Stenger, and Dominik Goeke. **The Electric Vehicle-Routing Problem with Time Windows and Recharging Stations**. *Transportation Science* 48:4, pages 500–520, 2014.
Cited on page 16.

[Sto12a] Sabine Storandt. **Quick and Energy-Efficient Routes: Computing Constrained Shortest Paths for Electric Vehicles**. In *Proceedings of the 5th ACM SIGSPATIAL International Workshop on Computational Transportation Science (IWCTS'12)*, pages 20–25. ACM, 2012.
Cited on pages 3, 4, 14, 15, 38, 77, 127, 174, 182.

[Sto12b] Sabine Storandt. **Route Planning for Bicycles – Exact Constrained Shortest Paths Made Practical via Contraction Hierarchy**. In *Proceedings of the 22nd International Conference on Automated Planning and Scheduling (ICAPS'12)*, pages 234–242. AAAI Press, 2012.
Cited on pages 12, 14, 255.

[Sto13] Sabine Storandt. **Algorithms for Vehicle Navigation**. PhD thesis. Universität Stuttgart, 2013.
Cited on pages 13, 38, 94, 95, 96.

[Sur86] Subhash Suri. **A Linear Time Algorithm for Minimum Link Paths Inside a Simple Polygon**. *Computer Vision, Graphics, and Image Processing* 35:1, pages 99–110, 1986.
Cited on pages 187, 191, 208, 209, 218, 245, 250.

[SV86] Robert Sedgewick and Jeffrey S. Vitter. **Shortest Paths in Euclidean Graphs**. *Algorithmica* 1:1, pages 31–48, 1986.
Cited on pages 10, 32.

[SW11] Peter Sanders and Dorothea Wagner. **Algorithm Engineering**. *it – Information Technology* 53:6, pages 263–265, 2011.
Cited on page 2.

[SW18] Maximilian Schiffer and Grit Walther. **An Adaptive Large Neighborhood Search for the Location-Routing Problem with Intra-Route Facilities**. *Transportation Science* 52:2, pages 331–352, 2018.
Cited on page 16.

[SW91] Bradley S. Stewart and Chelsea C. White. **Multiobjective A***. *Journal of the ACM* 38:4, pages 775–814, 1991.
Cited on page 12.

[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, pages 12:1–12:23, 2000.
Cited on pages 10, 41, 79, 193.

[SWZ02] Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. **Using Multi-Level Graphs for Timetable Information in Railway Systems**. In *Proceedings of the 4th Workshop on Algorithm Engineering & Experiments (ALENEX'02)*. Volume 2409 of Lecture Notes in Computer Science, pages 43–59. Springer, 2002.
Cited on pages 3, 10, 41, 79, 80, 99, 193.

[SZ16] Zhonghao Sun and Xingshe Zhou. **To Save Money or to Save Time: Intelligent Routing Design for Plug-In Hybrid Electric Vehicle**. *Transportation Research Part D: Transport and Environment* 43, pages 238–250, 2016.
Cited on pages 15, 255.

[TA16] Bezaye Tesfaye and Nikolaus Augsten. **Reachability Queries in Public Transport Networks**. In *Proceedings of the 28th GI-Workshop on Foundations of Databases (GvDB'16)*. Volume 1594 of CEUR Workshop Proceedings, pages 109–114. CEUR-WS.org, 2016.
Cited on page 17.

[Tar75] Robert E. Tarjan. **Efficiency of a Good but Not Linear Set Union Algorithm**. *Journal of the ACM* 22:2, pages 215–225, 1975.
Cited on page 224.

[TC92] Chi Tung Tung and Kim Lin Chew. **A Multicriteria Pareto-Optimal Path Algorithm**. *European Journal of Operational Research* 62:2, pages 203–209, 1992.
Cited on pages 118, 152.

[Tho04] Mikkel Thorup. **Integer Priority Queues with Decrease Key in Constant Time and the Single Source Shortest Paths Problem**. *Journal of Computer and System Sciences* 69:3, pages 330–353, 2004.
Cited on page 27.

[THS09] Edward Tate, Michael O. Harpster, and Peter J. Savagian. **The Electrification of the Automobile: From Conventional Hybrid, to Plug-In Hybrids, to Extended-Range Electric Vehicles**. *SAE International Journal of Passenger Cars – Electronic and Electrical Systems* 1:1, pages 156–166, 2009.
Cited on page 1.

[Tie+12] Tessa Tielert, David Rieger, Hannes Hartenstein, Raphael Luz, and Stefan Hausberger. **Can V2X Communication Help Electric Vehicles Save Energy?** In *Proceedings of the 12th International Conference on ITS Telecommunications (ITST'12)*, pages 232–237. IEEE, 2012.
Cited on pages 37, 175.

[Uhr+15] Martin Uhrig, Lennart Weiß, Michael Suriyah, and Thomas Leibfried. **E-Mobility in Car Parks – Guidelines for Charging Infrastructure Expansion Planning and Operation Based on Stochastic Simulations**. In *Proceedings of the 8th International Electric Vehicle Symposium and Exhibition (EVS28)*, 2015.
Cited on pages 105, 170.

[Wan91] Cao An Wang. **Finding Minimal Nested Polygons**. *BIT Numerical Mathematics* 31:2, pages 230–236, 1991.
Cited on pages 187, 191.

[Wid87] Peter Widmayer. **On Approximation Algorithms for Steiner's Problem in Graphs**. In *Proceedings of the 12th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'86)*. Volume 246 of Lecture Notes in Computer Science, pages 17–28. Springer, 1987.
Cited on page 223.

[Wil64] John W. J. Williams. **Algorithm 232: Heapsort**. *Communications of the ACM* 7:6, pages 347–348, 1964.
Cited on page 27.

[WJM13] Yan Wang, Jianmin Jiang, and Tingting Mu. **Context-Aware and Energy-Driven Route Optimization for Fully Electric Vehicles via Crowdsourcing**. *IEEE Transactions on Intelligent Transportation Systems* 14:3, pages 1331–1345, 2013.
Cited on pages 14, 16.

[WS88] Ady Wiernik and Micha Sharir. **Planar Realizations of Nonlinear Davenport-Schinzel Sequences by Segments**. *Discrete & Computational Geometry* 3:1, pages 15–47, 1988.
Cited on page 50.

[WWW86] Ying Fung Wu, Peter Widmayer, and Chak Kuen Wong Wong. **A Faster Approximation Algorithm for the Steiner Problem in Graphs**. *Acta Informatica* 23:2, pages 223–229, 1986.
Cited on page 223.

[XDP16] Yan Xu, Ramon Dalmau, and Xavier Prats. **Effects of Speed Reduction in Climb, Cruise and Descent Phases to Generate Linear Holding at No Extra Fuel Cost**. In *Proceedings of the 7th International Conference on Research in Air Transportation (ICRAT'16)*, 2016.
Cited on page 15.

[Yan+15] Hongming Yang, Songping Yang, Yan Xu, Erbao Cao, Mingyong Lai, and Zhaoyang Dong. **Electric Vehicle Route Optimization Considering Time-of-Use Electricity Price by Learnable Partheno-Genetic Algorithm**. *IEEE Transactions on Smart Grid* 6:2, pages 657–666, 2015.
Cited on page 16.

[Yao+13] Enjian Yao, Zhiqiang Yang, Yuanyuan Song, and Ting Zuo. **Comparison of Electric Vehicle's Energy Consumption Factors for Different Road Types**. *Discrete Dynamics in Nature and Society* 2013, 2013.
Cited on pages 14, 136, 138.

[Yen70] Jin Y. Yen. **An Algorithm for Finding Shortest Routes from All Source Nodes to a Given Destination in General Networks**. *Quarterly of Applied Mathematics* 27:4, pages 526–530, 1970.
Cited on page 12.

[YHZ16] Shi You, Junjie Hu, and Charalampos Ziras. **An Overview of Modeling Approaches Applied to Aggregation-Based Fleet Management and Integration of Plug-In Electric Vehicles**. *Energies* 9:11, pages 968:1–968:18, 2016.
Cited on page 9.

[Zie01] Mark Ziegelmann. **Constrained Shortest Paths and Related Problems**. PhD thesis. Universität des Saarlandes, 2001.
Cited on page 12.

[Zun14] Tobias Zündorf. **Electric Vehicle Routing with Realistic Recharging Models**. Master's thesis. Karlsruhe Institute of Technology, 2014.
Cited on page 132.

[ZY15] Rui Zhang and Enjian Yao. **Electric Vehicles' Energy Consumption Estimation with Real Driving Condition Data**. *Transportation Research Part D: Transport and Environment* 41, pages 177–187, 2015.
Cited on page 14.

List of Figures

List of Tables

List of Acronyms

ALT	A*, Landmarks, Triangle Inequality
APSP	All-Pairs Shortest Path
BDB-MLD	Bidirectional Distance-Bounding MLD
BFS	Breadth-First Search
BPE-MLD	Bidirectional Profile-Evaluating MLD
BSP	Bicriteria Shortest Path (Algorithm)
BSS	Battery Swapping Stations
CCH	Customizable Contraction Hierarchies
CD	Core-Dijkstra
CFP	Charging Function Propagating (Algorithm)
CH	Contraction Hierarchies
CHArge	CH, A*, Charging Stops
CHAsp	CH, A*, Adaptive Speeds
CP	Core-PHAST
CQ	Cost of Queries
CRP	Customizable Route Planning
CSP	Constrained Shortest Path
DAG	Directed Acyclic Graph
DFS	Depth-First Search
DN	Deleted Neighbors
DT	Distance Table
EA	Earliest Arrival
ED	Edge Difference
EV	Battery Electric Vehicle
EVD	EV Dijkstra
FIFO	First-In-First-Out

FMLP	Fast Minimum-Link Paths
FPTAS	Fully Polynomial-Time Approximation Scheme
GRASP	Graph Separators, Range, Shortest Path
HBEFA	Handbook of Emission Factors for Road Transport
HL	Hub-Based Labeling
isoCRP	Isochrone CRP
isoGRASP	Isochrone GRASP
isoPHAST	Isochrone PHAST
KIT	Karlsruhe Institute of Technology
kNN	*k*-Nearest Neighbors
MINE	Multimodal Incremental Network Expansion
MINEX	MINE with Vertex Expiration
MLD	Multilevel Dijkstra
NAMOA*	New Approach to Multi-Objective A*
NUMA	Non-Uniform Memory Access
OSM	OpenStreetMap
PHAST	PHAST Hardware-Accelerated Shortest Path Trees
PHEM	Passenger Car and Heavy Duty Emission Model
POI	Point of Interest
PUNCH	Partitioning Using Natural Cut Heuristics
reGRASP	Restricted GRASP
RP-CU	Range Polygon from Connected Unreachable Components
RP-RC	Range Polygon from Reachable Extracted Component
RP-SI	Range Polygon with Self-Intersections
RP-TS	Range Polygon from Triangular Separators
RPHAST	Restricted PHAST

SC	Shortcut Complexity
SoC	State of Charge
SPSP	Single-Pair Shortest Path
SRTM	Shuttle Radar Topography Mission
SSSP	Single-Source Shortest Path
TFP	Tradeoff Function Propagating (Algorithm)
TNR	Transit Node Routing
Uni-MLD	Unidirectional MLD

A Curriculum Vitæ

Name	Moritz Baum
Place of Birth	Eutin, Germany
Nationality	German

Education and Professional Experience

07/2017–04/2018	Research assistant at the Institute of Theoretical Informatics, Karlsruhe Institute of Technology (KIT)
11/2011–07/2017	PhD student at the Institute of Theoretical Informatics, Karlsruhe Institute of Technology (KIT) Advisors: Prof. Dr. Dorothea Wagner, Prof. Dr. Stefan Funke
11/2011–12/2011	Research stay at the University of Warwick, Coventry, UK
10/2008–09/2011	Master of Science in Computer Science Karlsruhe Institute of Technology (KIT)
10/2004–08/2008	Bachelor of Science in Computer Science University of Lübeck

Teaching Experience

10/2017–03/2018	Practical course “Algorithm Engineering”
10/2017–03/2018	Seminar “Route Planning”
04/2017–07/2017	Lecture “Algorithms for Route Planning”
04/2017–07/2017	Teaching assistant for “Energy Informatics 2”
10/2016–03/2017	Practical course “Algorithm Engineering”

04/2016–08/2016	Seminar “Graph Algorithms”
04/2016–07/2016	Lecture “Algorithms for Route Planning”
10/2015–03/2016	Practical course “Algorithm Engineering”
10/2015–03/2016	Seminar “Algorithm Design”
04/2015–07/2015	Lecture “Algorithms for Route Planning”
04/2015–07/2015	Proseminar “The P versus NP problem”
10/2014–03/2015	Practical course “Algorithm Engineering”
04/2014–07/2014	Lecture “Algorithms for Route Planning”
10/2013–03/2014	Practical course “Algorithm Engineering”
10/2012–03/2013	Practical course “Algorithm Engineering”
04/2012–07/2012	Proseminar “The P versus NP problem”

Supervised Theses

06/2017–11/2017	Patrick Niklaus, A Unified Framework for Electric Vehicle Routing (Master’s thesis)
07/2017–10/2017	Florian Grötschla, On the Complexity of Public Transit Profile Queries (Bachelor’s thesis)
05/2017–10/2017	Huyen Chau Nguyen, Engineering Multimodal Transit Route Planning (Master’s thesis)
07/2016–11/2016	Christian Bräuer, Optimale zeitabhängige Pausenplanung für LKW-Fahrer mit integrierter Parkplatzwahl (Bachelor’s thesis)
12/2015–03/2016	Oliver Thal, Zeitabhängige energieoptimale Routen für Elektrofahrzeuge (Bachelor’s thesis)
06/2015–12/2015	Philipp Glaser, Finding Attractive Routes for Bicycles and Pedelecs (Diploma thesis)
06/2015–10/2015	Jonas Sauer, Energy-Optimal Routes for Electric Vehicles with Charging Stops (Bachelor’s thesis)
01/2015–06/2015	Valentin Buchhold, Fast Computation of Isochrones in Road Networks (Master’s thesis)
12/2014–05/2015	Alexander Wirth, Algorithms for Contraction Hierarchies on Public Transit Networks (Diploma thesis)

11/2014–04/2015	Simeon Andreev, Consumption and Travel Time Profiles in Electric Vehicle Routing (Master's thesis)
06/2014–11/2014	Tobias Zündorf, Electric Vehicle Routing with Realistic Recharging Models (Master's thesis)
04/2014–08/2014	Alexandru Lesi, Energy-Optimal Routing with Turn Costs for Electric Vehicles (Bachelor's thesis)
01/2014–06/2014	Matthias Preis, Adaptive Routenplanung mit Hilfe eines Geocast Protokolls (Diploma thesis)
08/2013–11/2013	Janis Hamme, Customizable Route Planning in External Memory (Bachelor's thesis)
03/2013–08/2013	Qibai Zhu, Consideration of Toll Costs for Route Planning in Road Networks (Diploma thesis)
03/2013–06/2013	Lorenz Hübschle-Schneider, Speed–Consumption Trade-Off for Electric Vehicle Routing (Bachelor's thesis)
02/2013–05/2013	Sven Zühlsdorf, Efficient Calculation and Visualisation of Range Polygons (Bachelor's thesis)
01/2012–05/2012	Jörg D. Weisbarth, Shortest-Path Cover auf eingeschränkten Graphklassen (Bachelor's thesis)

B List of Publications

Journal Articles

[1] **Scalable Exact Visualization of Isocontours in Road Networks via Minimum-Link Paths**. *Journal of Computational Geometry* 9:1, pages 24–70, 2018. Joint work with Thomas Bläsius, Andreas Gemsa, Ignaz Rutter, and Franziska Wegner.

[2] **Towards Route Planning Algorithms for Electric Vehicles with Realistic Constraints**. *Computer Science – Research and Development* 31:1, pages 105–109, 2016. Joint work with Julian Dibbelt, Andreas Gemsa, and Dorothea Wagner.

[3] **On the Complexity of Partitioning Graphs for Arc-Flags**. *Journal of Graph Algorithms and Applications* 17:3, pages 265–299, 2013. Joint work with Reinhard Bauer, Ignaz Rutter, and Dorothea Wagner.

Articles in Conference Proceedings

[4] **Modeling and Engineering Constrained Shortest Path Algorithms for Battery Electric Vehicles**. In *Proceedings of the 25th Annual European Symposium on Algorithms (ESA'17)*. Volume 87 of Leibniz International Proceedings in Informatics (LIPIcs), pages 11:1–11:16. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017. Joint work with Julian Dibbelt, Dorothea Wagner, and Tobias Zündorf.

[5] **Time-Dependent Route Planning for Truck Drivers**. In *Proceedings of the 8th International Conference on Computational Logistics (ICCL'17)*. Volume 10572 of Lecture Notes in Computer Science, pages 110–126. Springer, 2017. Joint work with Alexander Kleff, Christian Bräuer, Frank Schulz, Valentin Buchhold, and Dorothea Wagner.

[6] **Consumption Profiles in Route Planning for Electric Vehicles: Theory and Applications**. In *Proceedings of the 16th International Symposium on Experimental Algorithms (SEA'17)*. Volume 75 of Leibniz International Proceedings in Informatics (LIPIcs), pages 19:1–19:18. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2017. Joint work with Jonas Sauer, Dorothea Wagner, and Tobias Zündorf.

[7] **Computing Minimum-Link Separating Polygons in Practice**. In *Proceedings of the 32nd European Workshop on Computational Geometry (EuroCG'16)*, 2016. Joint work with Thomas Bläsius, Andreas Gemsa, Ignaz Rutter, and Franziska Wegner.

[8] **Scalable Exact Visualization of Isocontours in Road Networks via Minimum-Link Paths**. In *Proceedings of the 24th Annual European Symposium on Algorithms (ESA'16)*. Volume 57 of Leibniz International Proceedings in Informatics (LIPIcs), pages 44:1–44:17. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. Joint work with Thomas Bläsius, Andreas Gemsa, Ignaz Rutter, and Franziska Wegner.

[9] **Fast Exact Computation of Isochrones in Road Networks**. In *Proceedings of the 15th International Symposium on Experimental Algorithms (SEA'16)*. Volume 9685 of Lecture Notes in Computer Science, pages 17–32. Springer, 2016. Joint work with Valentin Buchhold, Julian Dibbelt, and Dorothea Wagner.

[10] **Dynamic Time-Dependent Route Planning in Road Networks with User Preferences**. In *Proceedings of the 15th International Symposium on Experimental Algorithms (SEA'16)*. Volume 9685 of Lecture Notes in Computer Science, pages 33–49. Springer, 2016. Joint work with Julian Dibbelt, Thomas Pajor, and Dorothea Wagner.

[11] **Shortest Feasible Paths with Charging Stops for Battery Electric Vehicles**. In *Proceedings of the 23rd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'15)*, pages 44:1–44:10. ACM, 2015. Joint work with Julian Dibbelt, Andreas Gemsa, Dorothea Wagner, and Tobias Zündorf.

[12] **Speed-Consumption Tradeoff for Electric Vehicle Route Planning**. In *Proceedings of the 14th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'14)*. Volume 42 of OpenAccess Series in Informatics (OASIcs), pages 138–151. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2014. Joint work with Julian Dibbelt, Lorenz Hübschle-Schneider, Thomas Pajor, and Dorothea Wagner.

[13] **Energy-Optimal Routes for Electric Vehicles**. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'13)*, pages 54–63. ACM, 2013. Joint work with Julian Dibbelt, Thomas Pajor, and Dorothea Wagner.

[14] **On the Complexity of Partitioning Graphs for Arc-Flags**. In *Proceedings of the 12th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'12)*. Volume 25 of OpenAccess Series in Informatics (OASIcs), pages 71–82. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2012. Joint work with Reinhard Bauer, Ignaz Rutter, and Dorothea Wagner.

Technical Reports

[15] **Scalable Isocontour Visualization in Road Networks via Minimum-Link Paths**. Technical report abs/1602.01777. ArXiv e-prints, 2016. Joint work with Thomas Bläsius, Andreas Gemsa, Ignaz Rutter, and Franziska Wegner.

[16] **Fast Computation of Isochrones in Road Networks**. Technical report abs/1512.09090. ArXiv e-prints, 2015. Joint work with Valentin Buchhold, Julian Dibbelt, and Dorothea Wagner.

[17] **Dynamic Time-Dependent Route Planning in Road Networks with User Preferences**. Technical report abs/1512.09132. ArXiv e-prints, 2015. Joint work with Julian Dibbelt, Thomas Pajor, and Dorothea Wagner.

[18] **Energy-Optimal Routes for Electric Vehicles**. Technical report 2013-06. Faculty of Informatics, Karlsruhe Institute of Technology, 2013. Joint work with Julian Dibbelt, Thomas Pajor, and Dorothea Wagner.

C Deutsche Zusammenfassung

In den vergangenen Jahren wurde eine Vielzahl von Verfahren zur Routenplanung in Transportnetzwerken entwickelt, die die Berechnung optimaler Routen innerhalb kürzester Zeit ermöglichen. Im klassischen Anwendungsfall sind Nutzer dabei an *schnellsten* Verbindungen interessiert. Das typische (und zumeist einzige) Optimierungskriterium auf Straßennetzwerken ist daher die Fahrzeit auf einer Route. Der bekannte Algorithmus von Dijkstra löst das resultierende Problem optimal und in polynomieller Laufzeit, jedoch benötigt die Berechnung der Lösung selbst mit moderner Hardware oftmals mehrere Sekunden auf großen, realistischen Eingabeinstanzen. Um beispielsweise interaktive Anwendungen für die Routenplanung zu ermöglichen, verwenden *Beschleunigungstechniken* daher einen Vorberechnungsschritt auf den statischen Eingabedaten. Mit Hilfe der vorberechneten Informationen lassen sich dann meist beweisbar optimale Routen in der Praxis effizient (beispielsweise innerhalb weniger Millisekunden) berechnen. Die Techniken unterscheiden sich dabei in den Kriterien Vorberechnungsaufwand, Speicherbedarf, durchschnittliche Rechenzeit pro Anfrage, sowie Einfachheit der Implementierung.

Mit einem zunehmenden Anteil rein elektrisch betriebener Fahrzeuge ändern sich jedoch auch Anforderungen an Verfahren für die Routenplanung. Ein entscheidendes Kriterium ist hierbei die eingeschränkte Reichweite von Elektrofahrzeugen. Bedingt durch begrenzte Kapazitäten der Akkumulatoren, derzeit noch nicht flächendeckend verfügbare Ladestationen, sowie verhältnismäßig lange Ladezeiten wurde der Begriff der *Reichweitenangst* geprägt: Nutzer von Elektrofahrzeugen meiden längere Strecken aus Angst, ihr Ziel nicht erreichen zu können. Neben offensichtlichen Ansätzen zur Entschärfung dieser Situation, etwa der Verbesserung der Ladeinfrastruktur, bietet dieses veränderte Szenario eine Reihe von Herausforderungen und Möglichkeiten für Anwendungen im Bereich der Routenplanung.

Die vorliegende Arbeit befasst sich mit der Entwicklung effizienter Algorithmen zur Routenplanung von Elektrofahrzeugen, unter Berücksichtigung der oben genannten Aspekte. Sie folgt dem Paradigma des *Algorithm Engineering*, welches neben dem Entwurf und der theoretischen Analyse auch die Implementierung und experimentelle Evaluation der Verfahren in den Vordergrund stellt. Erkenntnisse aus der Evaluation stoßen dabei gegebenenfalls den Entwurf von Anpassungen an, sodass die vier Schritte zyklisch durchlaufen werden.

Die Arbeit ist in drei thematische Abschnitte gegliedert. Der erste Teil beschäftigt sich mit Algorithmen zur Berechnung von Routen, die den Energieverbrauch minimieren. Eine Besonderheit ist die Berücksichtigung von *Rekuperation*, also des

Wiederaufladens des Akkumulators während der Fahrt durch elektrisches Bremsen. Im zweiten Teil werden Erweiterungen des klassischen Problems *Constrained Shortest Path* betrachtet. Hierbei soll eine möglichst schnelle Route gefunden und gleichzeitig sichergestellt werden, dass das Ziel mit dem aktuellen Ladestand auch erreicht werden kann. Zusätzlich planen die vorgestellten Erweiterungen gegebenenfalls notwendige Ladestopps ein oder erhöhen die Reichweite durch Geschwindigkeitsanpassung. Im letzten Teil wird die Visualisierung der aktuellen Restreichweite auf einer Straßenkarte thematisiert. In der Arbeit werden effiziente und gleichzeitig präzise algorithmische Verfahren präsentiert. Diese berechnen zunächst den erreichbaren Teil des Netzwerks und anschließend eine geeignete Visualisierung der Reichweite.

Alle vorgestellten Verfahren werden auf realistischen Eingabedaten evaluiert, um ihre praktische Eignung zu demonstrieren. Die größte verwendete Instanz repräsentiert beispielsweise das Straßennetzwerk von Westeuropa mit etwa 20 Mio. Knoten und 50 Mio. Kanten. Im Folgenden werden die einzelnen Themen genauer beleuchtet.

Energieoptimale Routen. Der erste Abschnitt befasst sich mit der Berechnung von Routen, die den Energieverbrauch minimieren. Dadurch soll die Reichweite erhöht und gleichzeitig der Reichweitenangst entgegengewirkt werden. Gegenüber der klassischen Routenplanung, in der Straßennetzwerke als Graphen mit nichtnegativen Kantengewichten modelliert werden, ergeben sich dabei verschiedene neue Herausforderungen. Die Möglichkeit zur Rekuperation etwa führt zu negativen Kantengewichten. Eine Variante von Dijkstras Algorithmus findet dann zwar nach wie vor die optimale Lösung, benötigt jedoch im schlimmsten Fall exponentielle Laufzeit. Zusätzlich müssen *Battery Constraints* berücksichtigt werden: Der Akkumulator darf zu keinem Zeitpunkt vollständig entladen werden und kann durch Rekuperation höchstens bis zur gegebenen Kapazität aufgeladen werden. Außerdem hängt der Energieverbrauch des Fahrzeugs in der Realität von einer Vielzahl von Parametern ab, wie etwa der Fahrweise, den aktuellen Wetterbedingungen und der Verkehrslage. Beschleunigungstechniken sollten daher benutzerspezifische Verbrauchsprofile erlauben und zudem deren regelmäßige Aktualisierung ermöglichen.

Verschiedene Ansätze zur effizienten Handhabung von negativen Verbrauchswerten werden vorgestellt und miteinander verglichen. Ein wesentlicher Bestandteil von Lösungsverfahren für die oben genannte Problemstellung ist die Berechnung von *Verbrauchsfunktionen*, die den Energieverbrauch auf einer Route als Funktion des Ladestands am Startknoten abbilden, um Battery Constraints zu modellieren. In dieser Arbeit wird gezeigt, dass die Beschreibungskomplexität solcher Funktionen linear in der Eingabegröße beschränkt ist. Mit Hilfe dieser Erkenntnis lassen sich Polynomialzeitverfahren für verschiedene erweiterte Problemstellungen ableiten, etwa die Berechnung von optimalen Routen für jeden möglichen Ladestand oder von verbrauchsminimalen Routen mit Ladestopps.

Zudem werden in der Arbeit verschiedene praktische Implementierungen der theoretisch untersuchten Verfahren vorgestellt und evaluiert. Insbesondere werden Ansätze vorgestellt, die mit Hilfe geeigneter Vorberechnung energieoptimale Routen innerhalb weniger Millisekunden berechnen. Um gleichzeitig die schnelle Integration neuer Verbrauchsfunktionen zu ermöglichen, wird eine auf dem bekannten Verfahren *Multilevel Dijkstra* basierende Technik vorgeschlagen, die um die oben genannten Funktionalitäten erweitert wird. Die praktische Eignung des resultierenden Verfahrens wird auf realistischen Eingabedaten demonstriert. Hierzu werden Verbrauchsdaten mit Hilfe eines detaillierten Fahrzeugmodells generiert. Es zeigt sich, dass das neue Verfahren bestehende Ansätze in der Laufzeit um mindestens eine Größenordnung verbessert und damit beispielsweise interaktive Anwendungen ermöglicht. Auch auf der größten Instanz (Westeuropa) liegen Rechenzeiten selbst für schwierige Anfragen im einstelligen Millisekundenbereich. Zudem lassen sich aktualisierte Verbrauchsfunktionen für das vollständige Netzwerk innerhalb weniger Sekunden integrieren.

Schnellstmögliche durchführbare Routen. Da eine sehr energiesparende Fahrweise meist mit signifikantem Zeitverlust einhergeht, liegt der Fokus im zweiten Teil der Arbeit auf der Berechnung klassischer schnellster Routen. Gleichzeitig soll jedoch die Durchführbarkeit der Route sichergestellt sein. Der Ladestand muss also ausreichen, um das Ziel auf der vorgeschlagenen Route zu erreichen. Es werden zwei Verallgemeinerungen des aus der Literatur bekannten $\mathcal{NP}$-schweren Problems *Constrained Shortest Path* betrachtet.

Kann das Ziel auf der schnellsten Route wegen eines zu hohen Energieverbrauchs nicht erreicht werden, kann die Reichweite nicht nur durch Ausweichen auf eine alternative Route, sondern auch durch Anpassung der Fahrweise erhöht werden – indem zum Beispiel auf schnellen Strecken die Geschwindigkeit bewusst reduziert wird. Im zugrunde liegenden Modell ergibt sich damit für einzelne Streckensegmente statt skalarer Kantengewichte ein funktionaler Zusammenhang zwischen Reisezeit und Energieverbrauch. Es wird gezeigt, dass sich Verfahren zur Lösung von *Constrained Shortest Path* auf das Propagieren derartiger Funktionen erweitern lassen. Die Anpassung von Beschleunigungstechniken auf dieses Problem gestaltet sich dagegen besonders schwierig: Da der Energieverbrauch auf einer Route sowohl von der gewählten Geschwindigkeit als auch vom Ladestand abhängig ist, müssen diese Techniken beide Dimensionen auf den Energieverbrauch abbilden. In der Arbeit wird ein Ansatz basierend auf der Technik *Contraction Hierarchies* vorgestellt, der diese Abhängigkeit mit Hilfe univariater Funktionen modelliert. In Verbindung mit Varianten des A^*-Algorithmus lassen sich damit für typische Reichweiten (bis zu mehreren hundert Kilometern) optimale Lösungen in der Praxis schnell berechnen.

Ein weiterer wichtiger Aspekt bei der Routenplanung ist die Berücksichtigung von Ladestopps. Auch dieses Szenario ist in der Realität sehr komplex, denn die Ladedauer beeinflusst sowohl die verfügbare Restreichweite als auch die Gesamtreisezeit. Der Ladevorgang ist zudem je nach Ladestationstyp unterschiedlich und typischerweise nichtlinear (der Ladestrom sinkt mit zunehmendem Ladestand, somit steigt die Ladedauer). Ein Lösungsverfahren muss all diese Bedingungen berücksichtigen, um in der Praxis verwendbare Ergebnisse zu liefern. Auch für diese Problemstellung wird gezeigt, dass es durch Erweiterung grundlegender Algorithmen für bikriterielle Suche in Exponentialzeit optimal gelöst werden kann. Zudem werden Beschleunigungstechniken basierend auf Contraction Hierarchies und A^*-Suche präsentiert.

Alle vorgestellten Verfahren werden einer umfangreichen experimentellen Auswertung unterzogen. Für beide betrachtete Problemstellungen zeigt sich, dass mit Hilfe der vorgestellten Beschleunigungstechniken trotz theoretisch exponentieller Laufzeit optimale Lösungen für realistische Anfrageszenarien innerhalb von Sekunden berechnet werden können. Mit Hilfe heuristischer Erweiterungen können zudem Lösungen mit kleinem Fehler bei gleichzeitig wesentlich geringerem Rechenaufwand ermittelt werden, wodurch Laufzeiten im zweistelligen Millisekundenbereich und somit interaktive Anwendungen ermöglicht werden.

Visualisierung der Restreichweite. Der letzte Abschnitt der Arbeit beschäftigt sich mit der Visualisierung der aktuellen Restreichweite, einem weiteren wichtigen Aspekt zur Minderung der Reichweitenangst. Ein entsprechendes Verfahren muss zwei Teilprobleme möglichst effizient lösen: Zunächst muss der mit aktuellem Ladestand erreichbare Teil des Straßennetzwerks identifiziert werden, anschließend ist dieser geeignet visuell darzustellen. Beide Schritte sollen in der Praxis schnell genug sein, um interaktive Anwendungen zu unterstützen.

Das erste Teilproblem ist somit die effiziente Berechnung des erreichbaren Subnetzwerks. Hierzu wird zunächst eine kompakte Repräsentation dieses Subnetzwerks vorgeschlagen. Diese lässt sich mit einer Variante von Dijkstra's Algorithmus in Polynomialzeit berechnen, die aber verhältnismäßig rechenintensiv ist. Für praktikablere Laufzeiten werden verschiedene Beschleunigungstechniken auf die gegebene Problemstellung erweitert. Die verschiedenen Techniken unterscheiden sich dabei in den Kriterien Vorberechnungsaufwand, Speicherbedarf und Anfragezeit. Somit ergibt sich ein Portfolio an Verfahren, die sich für unterschiedliche Anwendungsfälle eignen. Alle Verfahren ermöglichen Anfragezeiten von weniger als 100 ms (Westeuropa), selbst für hohe Reichweiten. Zudem lassen sich die Ansätze parallelisieren, wodurch Laufzeiten von deutlich unter 10 ms erreicht werden können.

Der zweite Schritt beinhaltet die eigentliche Darstellung der Reichweite durch ein Polygon, das den erreichbaren Teil des Straßennetzwerks vom restlichen, unerreichbaren Teil trennt. Neben praktikabler Laufzeit sind die wesentlichen Ziele bei dieser Problemstellung die Exaktheit der Lösung (erreichbare und unerreichbare Teile des Netzwerks sollen korrekt separiert werden), sowie die Minimierung der Anzahl der Segmente des berechneten Polygons (um beispielsweise effizientes Rendering zu ermöglichen). Es ist nicht bekannt, ob das resultierende Problem in Polynomialzeit lösbar ist. Zudem hat selbst für eine vereinfachte Variante der beste bekannte Algorithmus quadratische Laufzeit. In dieser Arbeit werden stattdessen Linearzeitalgorithmen vorgestellt, die die Exaktheit der Lösung garantieren und die Anzahl der Segmente heuristisch minimieren. Ein wichtiger Bestandteil aller Ansätze ist ein neuer Algorithmus zur Berechnung von *Minimum Link Paths* in einfachen Polygonen, also von Polygonzügen innerhalb gegebener Polygone mit minimaler Anzahl Segmente. Für dieses in der Theorie gut untersuchte Problem wird ein neuer Linearzeitalgorithmus vorgestellt, der wesentlich einfacher ist als bestehende Verfahren. Damit lassen sich für den zweiten Schritt exakte und in der Anzahl verwendeter Segmente nahezu optimale Lösungen in deutlich weniger als 100 ms (Westeuropa) berechnen.