Algorithmica (2020) 82:1490–1546
https://doi.org/10.1007/s00453-019-00655-9

Energy-Optimal Routes for Battery Electric Vehicles

Moritz Baum[1] · Julian Dibbelt[2] · Thomas Pajor[2] · Jonas Sauer[1] · Dorothea Wagner[1] · Tobias Zündorf[1]

Received: 17 May 2018 / Accepted: 14 November 2019 / Published online: 3 December 2019

Abstract

We study the problem of computing paths that minimize energy consumption of a battery electric vehicle. For that, we must cope with specific properties, such as regenerative braking and constraints imposed by the battery capacity. These restrictions can be captured by *profiles*, which are a functional representation of optimal energy consumption between two locations, subject to initial state of charge. Efficient computation of profiles is a relevant problem on its own, but also a fundamental ingredient to many route planning approaches for battery electric vehicles. In this work, we prove that profiles have linear complexity. We examine different variants of Dijkstra's algorithm to compute energy-optimal paths or profiles. Further, we derive a *polynomial-time* algorithm for the problem of finding an energy-optimal path between two locations that allows stops at charging stations. We also discuss a heuristic variant that is easy to implement, and carefully integrate it with the well-known Contraction Hierarchies algorithm and A* search. Finally, we propose a practical approach that enables computation of energy-optimal routes within milliseconds after fast (metric-dependent) preprocessing of the *whole* network. This enables flexible updates due to, e. g., weather forecasts or refinements of the consumption model. Practicality of our approaches is demonstrated in a comprehensive experimental study on realistic, large-scale road networks.

Keywords Algorithm engineering · Shortest paths · Speedup techniques · Electric vehicles · Profile search

Preliminary versions of this manuscript have appeared in the proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems [10] and the 16th International Symposium on Experimental Algorithms [13]. It is based on the thesis of one of the authors [7].
Tobias Zündorf acknowledges support by DFG Research Grant WA 654/23-1.

Extended author information available on the last page of the article

1 Introduction

Route planning services explicitly designed for electric vehicles (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*, the fear of getting stranded due to insufficient range. 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 [29,51]. As a result, energy consumption can become *negative* on some road segments (to model recuperation). Moreover, 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 [29,55]. 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 [29]. In this work, we examine the complexity of profiles in road networks.

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 [61]. Therefore, we also discuss approaches that explicitly consider stops at charging stations. Even when optimizing for energy-consumption only, integrating charging stops into route planning is nontrivial: 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.

Altogether, 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 work are relevant for closely related multicriteria scenarios, where the additional consideration of overall trip time in route optimization yields complex problem settings [8,59,62].

Related Work Classic route planning approaches apply Dijkstra's algorithm [26] to a graph representation of the network, with fixed scalar edge costs representing, e. g., travel time. For faster running times in practice, *speedup techniques* [3] accelerate online shortest-path queries with data *preprocessed* in an offline phase. Examples of such techniques are Contraction Hierarchies (CH) [33], where vertices are contracted iteratively and replaced by *shortcuts* in the graph, and variants of A* search [34,37]. Combining both techniques, Core-ALT [6] contracts most vertices and runs A* on the remaining *core graph*. Some techniques were also extended to more complex scenarios, such as time-dependent cost functions that model, e. g., traffic during peak hours [5,11,19,23]. In this context, a *profile query* asks for a functional representation of travel time between locations for *any* departure time. Such functions may have superpolynomial complexity [32], but can be computed by an output-sensitive search algorithm [18,24]. To enable relatively fast integration traffic updates or user

preferences, more recent techniques allow an additional *customization* phase after preprocessing, which enables fast updates of the whole network to quickly incorporate global traffic updates or user preferences [20,25,28]. See Bast et al. [3] for a more complete survey.

Regarding route planning for EVs, energy consumption may induce negative cost 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 [41]. The well-known algorithm of Bellman–Ford [14,31] handles negative edge costs and has quadratic time complexity on sparse networks. To get rid of negative edge costs, one can use a technique called *potential shifting* [42]. This reenables Dijkstra's (label-setting) algorithm.

Artmeier et al. [1] handle negative costs with the Bellman–Ford 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 moderate scale (representing Bavaria, Germany) in their implementation. Constraints imposed by the battery capacity are checked explicitly in the algorithm during edge relaxation, without affecting its asymptotic complexity. Using a simple physical energy consumption model, Sachenbacher et al. [51] combine potential shifting (obtained directly from the consumption model) with A* search to get a factor of 3 speedup over their previous label-correcting approach.

Eisner et al. [29,60] 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 [4,24,27], but mapping SoC to energy consumption. They show that the complexity of the consumption profile of a path is constant. Further, they adapt CH to compute optimal routes in less than 100 ms on large graphs, making their technique the fastest one available for the problem of computing energy-optimal routes.

Integrating battery constraints into route planning via piecewise linear functions, Schönfelder et al. [55] consider *profile search* to compute optimal consumption for *every* initial SoC between a given pair of vertices, along the lines of time-dependent profiles [18,24]. Variants of A* search and CH are proposed, the latter of which has average running times of a few milliseconds on a relatively small road network (representing parts of Bavaria, Germany). 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 [54].

Kluge et al. [44] 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.

Stops at charging stations are often considered under the simplifying assumptions that the charging always results in a *fully* recharged battery [35,46,58,59,61,63]. Routes with a minimum number of intermediate charging stops can then be computed in less than a second on road networks of moderate size [59,61]. More complex models also consider constraints on time spent driving [9,12,44] and recharging [8,40,45,47,64,67],

but this results in much more difficult (typically $\mathcal{NP}$-hard) problems and the proposed techniques are either inexact or impractical.

Contribution and Outline In Sect. 2, we formally introduce our model of battery constraints and state two problem variants. The first asks for an energy-optimal route for a given initial SoC, whereas the second requires a *profile*, i. e., an energy-optimal route for *every* initial SoC. We examine the complexity of profiles and, as our main result, prove that they have linear complexity—much in contrast to profiles in time-dependent routing, which can have superpolynomial size [32]. We also derive basic operations to concatenate and merge profiles.

We investigate approaches based on Dijkstra's algorithm to compute energy-optimal routes and profiles in Sect. 3. Aiming at practical solutions, we explore different strategies to handle recuperation (i. e., negative costs). By presenting a polynomial-time algorithm to compute profiles, we efficiently solve a problem that is not only relevant on its own, but is a crucial ingredient of speedup techniques in our scenario.

In Sect. 4, we consider energy-optimal routes that allow stops at charging stations to recharge the battery. Unlike previous studies [35,58,59], 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. Building upon our theoretical findings, we derive a *polynomial-time* algorithm to solve the problem. To make the approach practical, we propose a (heuristic) variant and integrate it with CH [29,33] and A* search [37].

In Sect. 5, 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 Customizable Route Planning (CRP) method of Delling et al. [20] to handle battery constraints and achieve fast (metric-dependent) preprocessing. We propose several query algorithms to compute energy-optimal routes.

Section 6 presents our experimental results on large, realistic road networks. It demonstrates the excellent performance of our algorithms in practice: Even for long-range queries across Europe we achieve query times of well below a second in the most difficult setting considered. Altogether, our algorithms exhibit faster query times than previous approaches, while improving preprocessing time by up to three orders of magnitude. We conclude with final remarks in Sect. 7.

2 Integrating Battery Constraints

In what follows, we 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 (Sect. 2.1). 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 (Sect. 2.2).

2.1 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 [1,29]. 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 ∞ 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.

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 Sect. 4, 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 (finite) *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 section, 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 predecessor pointers, as in Dijkstra's algorithm [16].

2.2 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 (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*. Afterwards, we consider the general scenario, where multiple paths may contribute to the same profile. It turns out that the function is *piecewise linear* in each case and can be represented as follows. 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 $x_i \leq x_j$ for $i < j$, $f(b) = -\infty$ for $b < x_1$, $f_P(b) = y_k$ for $b \geq x_k$, and the function is evaluated by linear interpolation between two consecutive breakpoints for $b \in [x_1, x_k)$. Note that we allow the case $x_i = x_{i+1}$ for two consecutive breakpoints to model discontinuities. An empty sequence $F = \emptyset$ represents the function $f \equiv -\infty$.

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

$$f_{(u,v)}(b) := \begin{cases} b - a^+ & \text{if } b \geq a^+, \\ -\infty & \text{otherwise.} \end{cases} \tag{1}$$

The function $f_{(u,v)}$ is represented by the sequence $F_{(u,v)} = [(a^+, 0), (M, M - a^+)]$ consisting of two breakpoints; see Fig. 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

$$f_{(u,v)}(b) := \begin{cases} b - a^- & \text{if } b - a^- \leq M, \\ M & \text{otherwise.} \end{cases} \tag{2}$$

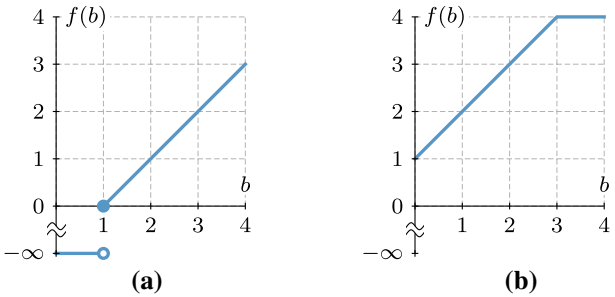

Fig. 1 SoC functions for different edge costs, assuming a battery capacity of $M = 4$. **a** The SoC function of an edge with cost 1. **b** The SoC function of an edge with cost -1

Again, the SoC function is represented by a sequence consisting of two breakpoints, namely $F_{(u,v)} = [(0, -a^-), (M + a^-, M)]$. Figure 1b shows an SoC function that represents a single edge with negative cost.

Profiles Representing Paths Eisner et al. [29] 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 1. Additionally, Lemma 1 provides a general specification of SoC functions for single paths.

Before proving Lemma 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. (We define the cost of a path 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 Fig. 2. The following Lemma 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 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 is a subpath of P with cost greater than M, $F_P = \emptyset$ and $f_P \equiv -\infty$.*
2. *Otherwise, if some subpath has cost below $-M$, $F_P = [(c(P_s^+), M - c(P_t^+))]$.*
3. *If neither 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 $v \in V$.

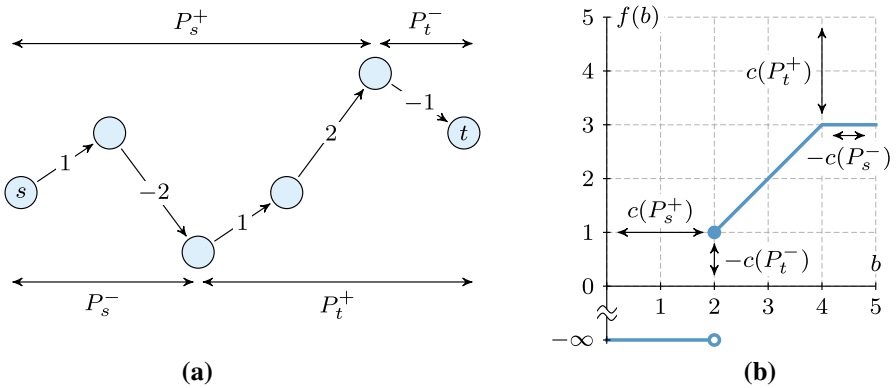

Fig. 2 An s–t path together with its SoC function, assuming that the battery capacity is $M = 5$. **a** The s–t path with depicted edge costs. The cost of the path is 1 and its important subpaths are indicated. Relative vertical positions of vertices correspond to costs of subpaths starting or ending 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

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 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. □

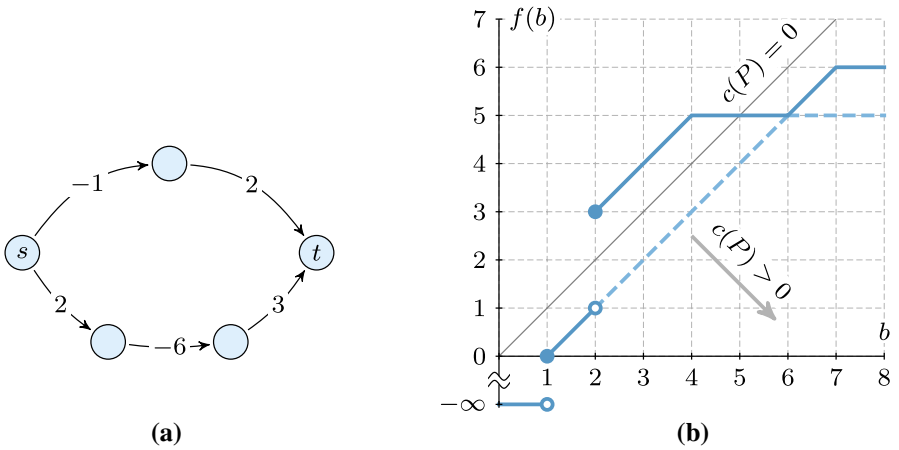

Fig. 3 The SoC profile of two vertices in a graph. The battery capacity is $M = 8$. **a** The graph with indicated source s and target t. There are two different s–t paths with respective costs 1 and -1. **b** The corresponding SoC function. The dashed segments indicate dominated parts of SoC functions 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 1)

According to Lemma 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 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 Fig. 2.

In summary, at most two breakpoints are necessary to represent the SoC function of a path. This stands much 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 [4,11,24].

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 Fig. 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 Fig. 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 2 is a direct implication of the observations by Atallah [2, Lemma 2.2] and Davenport

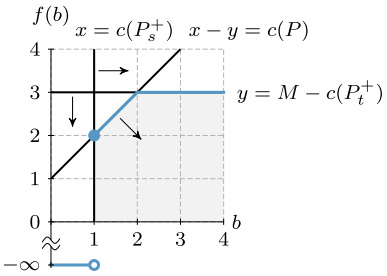

Fig. 4 Dominated area of an SoC function, for $M = 4$ and a path P with $c(P) = -1$. Its important subpaths have cost $c(P_s^+) = 1$ and $c(P_t^+) = 1$. The costs induce three lines, each of which subdivides the Euclidean plane into two half planes. The SoC function of a path Q with $c(Q) \geq c(P)$,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

and Schinzel [17, Theorem 1]; see also the introduction in Wiernik and Sharir [68]. (Note that the number of breakpoints in the upper envelope of linear functions can be superlinear in general [68].)

Lemma 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 $|\mathcal{P}|$.*

Since the number of s–t paths can be exponential in the graph size, Lemma 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*.

We continue with some basic properties of paths and their SoC functions. First, Lemma 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 1 and is illustrated in Fig. 4.

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

Lemma 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 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 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 $\mathcal{O}(|V|^2)$ paths contributing to an s–t profile. We formally prove the claim in Lemma 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 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, 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 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 1 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 Fig. 5. Hence, the bound of Theorem 1 is tight up to a constant factor and the number of breakpoints of an SoC function is in $\Theta(|V|)$ in the worst case.

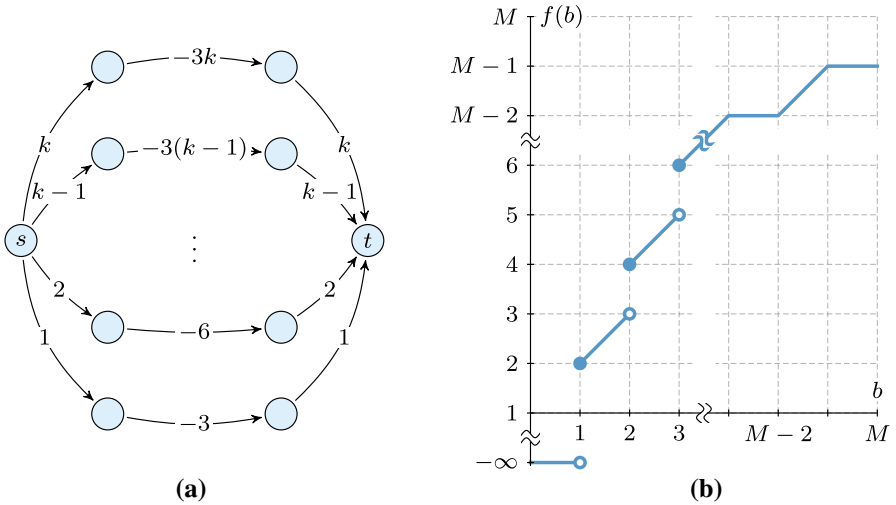

Fig. 5 An SoC function with $\Theta(|V|)$ breakpoints. **a** The input graph with designated vertices s and t. There are $k \in \mathbb{N}$ distinct s–t paths and $|V| = 2(k+1)$. Edges are labeled with their costs. **b** A sketch of the s–t profile for an arbitrary battery capacity $M \geq 3k$. Every 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 $2(|V| - 3)$ breakpoints in total

Theorem 1 *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 $\mathcal{O}(|V|)$.*

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 $\mathcal{O}(|V|)$. Together with Lemma 2, this proves our claim.

The undirected graph G' that 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 function: 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 5 implies that there are no multi-edges in the resulting graph. By construction, G' consists of at least two components and each component induces a bipartite subgraph. We claim that G' contains no cycles. This implies that G' has at most $\mathcal{O}(|V|)$ edges, which proves the theorem.

Assume for contradiction that there is a 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 $V_1^- \cup V_2^+$.

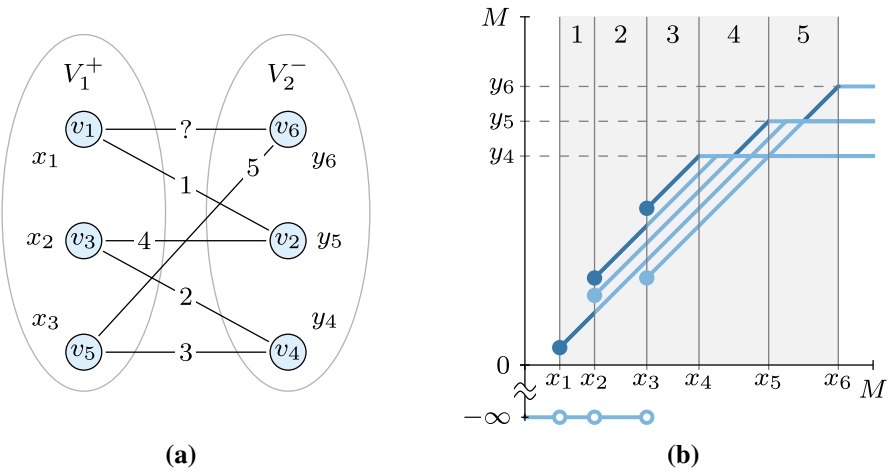

Fig. 6 Illustration of the proof of Theorem 1. **a** A constructed bipartite graph G' with copies of top and bottom vertices of the input graph. Edges represent paths connecting certain important vertices. Vertices have assigned real-valued constants $x_1, x_2, x_3, y_4, y_5, y_6$. Edge labels indicate the interval in **b** where the corresponding characteristic segment is contained in the upper envelope of all functions. **b** The SoC functions of 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 function (Color figure online)

Case 1 All edges represent top-bottom paths, and therefore $\{v_1, \ldots, v_k\} \subseteq V_1^+ \cup V_2^-$. Figure 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 implies that these paths also share the same maximum prefix with some length $x \in [0, M]$. Therefore, by Lemma 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 Fig. 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 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, as shown in Fig. 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 Fig. 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, these paths 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 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 $x \in [0, M]$. Along the lines of the first case, this yields a contradiction. □

3 Basic Algorithms

In this section, we discuss basic algorithms to answer SoC queries (Sect. 3.1) and profile queries (Sect. 3.2) 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 [29]. This important property enables label-*setting* algorithms (i. e., algorithms that visit each vertex at most once).

3.1 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 algorithm for such queries is a known (label-correcting) variant of Dijkstra's algorithm [1,41], which we refer to as *EV Dijkstra (EVD)*. See Algorithm 1 for pseudocode. Along the lines of Dijkstra's algorithm [26], 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 Eqs. 1 and 2 in Sect. 2.2 that

Algorithm 1: EV Dijkstra.

```
Input: A graph G = (V, E), a cost function c: E → ℝ, a source vertex s ∈ V, a battery
       capacity M ∈ ℝ≥0, and an initial SoC b_s ∈ [0, M].
Output: For every vertex v ∈ V, the algorithm computes the optimal SoC upon arrival at v.
// initialize labels
1  foreach v ∈ V do
2      b(v) ⟵ −∞;
3  b(s) ⟵ b_s;
4  Q.insert(s, b_s)
// run main loop
5  while Q.isNotEmpty() do
6      u ⟵ Q.deleteMax();
7      foreach (u, v) ∈ E do
8          b ⟵ f_(u,v)(b(u));
9          if b > b(v) then
10             b(v) ⟵ b;
               // insert, reinsert, or update the vertex
11             Q.update(v, b);
```

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

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 might as well propagate labels with *minimum* energy consumption.) Correctness follows from the FIFO property [27] and the fact that the above variant of Dijkstra's algorithm correctly handles negative costs [41].

If edges with negative cost exist, the algorithm is label *correcting*: It may scan and reinsert vertices into the queue 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 [41]. 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.

We now discuss how the performance of EVD can be improved by aborting the search early if a shortest path was already found (at the cost of some preprocessing effort). Including techniques from the literature [1,29,51], we provide an overview of ways to achieve this. See Sect. 6.2 for an experimental comparison of preprocessing time, space consumption, and query performance of the different approaches presented below.

Enabling a Stopping Criterion The performance of Dijkstra's algorithm can be improved by making use of the *stopping criterion*: The algorithm can terminate as soon as the target vertex is extracted from the queue. For EVD, since it is label correcting, this cannot be applied: After the target 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 ways to establish a stopping criterion for EVD.

Given the target t, let P_t^* denote the shortest v–t path in G from any vertex $v \in V$, with cost $c_t^* := c(P_t^*)$. 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 $|V|$ 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. For each $v \in V$, the length of the shortest s'–v path equals the value c_v^*. To compute these paths, we can use Dijkstra's algorithm [26], which loses its label-setting property in the presence of edges with negative costs [41]. Although it has exponential worst-case running time, it outperforms the polynomial-time algorithm of Bellman–Ford [14,31] on realistic instances [1]. 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 setting all vertex labels to 0. 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 its distance label still equals 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 [42]. They allow us to apply the regular stopping criterion, i. e., the search is aborted once the target is scanned. Recall that vertex labels in EVD rep-

resent 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 (because this implies that a label can never be improved after it was extracted from the queue). Hence, we *subtract* potentials from keys when updating labels and a consistent potential function $\pi : V \to \mathbb{R}$ 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 π should fulfill the condition $c(u, v) - \pi(u) + \pi(v) \geq 0$. If this is the case, π is also called *consistent*. Note that any consistent potential induces a nonnegative *reduced edge cost function* c' by *shifting* the cost of every edge $(u, v) \in E$ by its potential difference, setting $c'(u, v) := c(u, v) - \pi(u) + \pi(v)$. We discuss different ways to obtain consistent potential functions, which make EVD label setting [29].

The first variant, also proposed by Eisner et al. [29], makes use of a *vertex-induced* potential function $\pi_v : V \to \mathbb{R}$. It takes an arbitrary fixed vertex $v \in V$ and sets $\pi_v(u) := \text{dist}_c(u, v)$ for all vertices $u \in V$, i. e., the distance from u to v with respect to the cost function c (ignoring battery constraints). Computing these values requires a single run of Dijkstra's (label-correcting) algorithm on the *backward* graph $\bar{G}$ (defined as the graph G with all edge directions inverted) with cost function c (which carries over to the backward graph canonically). Consistency of π_v follows immediately from the triangle inequality.

The second variant uses of a *bound-induced* potential function $\pi_b : V \to \mathbb{R}$, 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 [15], which preliminary experiments indicated to be slower in our setting.

Finally, we propose a *height-induced* potential $\pi_h : V \to \mathbb{R}$. 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 [29,51]. 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 difference $\Delta(e)$.

It is easy to construct examples where $\underline{\alpha} > \bar{\alpha}$ 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 $c(e) = \ell(\lambda_1 + s\lambda_2)$ in common physical models [30,51], 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

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

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.

3.2 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 Sect. 2.2, charging more than that might even enable paths with lower energy consumption (such as paths of lower overall cost that first go uphill). Since 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 that we introduce in the next sections.

Generally, a profile search replaces scalar edge costs by functions of some variable, which represents a certain state at the tail vertex of the edge [24,54]. Time-dependent route planning is probably the most well-known example, where edge costs represent travel times depending on the current point in time to account for, e. g., peak and off-peak hours [4,24,32]. Although computationally more expensive, profile search is conceptually easy: One can extend Dijkstra's algorithm [24,50], which we now adapt to our setting.

First, 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

Algorithm 2: Label-correcting profile search.

```
Input: A graph G = (V, E), a cost function c: E → ℝ, a source vertex s ∈ V, and a battery
       capacity M ∈ ℝ≥0.
Output: An s–v profile for every vertex v ∈ V.
// initialize labels
foreach v ∈ V do
    f_v ⟵ f_−∞;
f_s ⟵ id;
Q.insert(s, 0);
// run main loop
while Q.isNotEmpty() do
    u ⟵ Q.deleteMin();
    foreach (u, v) ∈ E do
        f ⟵ link(f_u, f_(u,v));
        if ∃b ∈ [0, M]: f(b) > f_v(b) then
            f_v ⟵ merge(f_v, f);
            Q.update(v, key(f_v));
```

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 link: $\mathbb{F} \times \mathbb{F} \to \mathbb{F}$ 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 merge: $\mathbb{F} \times \mathbb{F} \to \mathbb{F}$ 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 $b \in [0, M]$. 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. Furthermore, our algorithm requires *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 $\text{merge}(f_1, f_2) = f_1$.

Our profile search is outlined in Algorithm 2. Starting from the source vertex $s \in V$, it 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 Fig. 3 from Sect. 2.2. The algorithm initializes $f_v \equiv -\infty$ (denoted $f_{\{-\infty\}}$ in Algorithm 2) for all $v \in V$ except s, for which the SoC function $f_s = \text{id}$ 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 Fig. 3b). In each step of the main loop, the algorithm scans a vertex $u \in V$ with minimum key and follows Dijkstra's algorithm. When scanning an edge $(u, v) \in E$, the profiles f_u and $f_{(u,v)}$ 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.

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 Sect. 3.1 to remedy this issue. Then, we can apply the following *target pruning* rule, which is applied in combination with vertex potentials or bounds induced by values c_v^* for all $v \in V$, as introduced in Sect. 3.1. 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 $c_v^{\max \le M} \in [0, M] \cup \{\infty\}$ 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 $c_v^{\max \le M} := \infty$ if $f_v \equiv -\infty$). Then, whenever the algorithm scans a vertex $v \in V$, it checks whether both $b_v^{\min} \ge b_t^{\min}$ and $c_v^{\min} \ge c_t^{\max \le M}$ hold, where $c_v^{\min}$ is the minimum energy consumption of the current profile at v. If that is the case, the algorithm *prunes* the search at the vertex v, i. e., it does not scan any outgoing edges of v.

Implementation of Linking and Merging 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 Sect. 2.2, 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 Sect. 2.2) 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 Sect. 2.2, 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 $a_1 \ge 0$ and $a_2 \ge 0$ hold or $a_1 \le 0$ and $a_2 \le 0$ 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 $\min\{a_1, a_2\}$.

We showed in Sect. 2.2 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 approaches for time-dependent route planning [24]. 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 $y \geq f_2(x)$ and vice versa. Additional breakpoints are necessary at intersections of both functions, while unnecessary collinear points can be removed.

As an optimization, our implementation uses 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). 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. 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 a significant amount of space and about a factor of 2 in running time.

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 $\pi : V \to \mathbb{R}$ 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 $\mathcal{O}(|V|^2)$. 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 Theorem 2.

Theorem 2 *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.*

4 Energy-Optimal Routes with Charging Stops

In the previous section, we have 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.

We formally describe how we extend our model and define the problem (Sect. 4.1), before we introduce a basic label-setting algorithm to solve it (Sect. 4.2). We discuss a conceptually simple polynomial-time approach (Sect. 4.3) and propose a framework to implement it efficiently (Sect. 4.4).

4.1 Model and Problem Statement

In addition to our previous setting (see Sect. 2.1), 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 [29]. 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 battery swapping stations (stations at which the battery is immediately replaced with a fully charged one), 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 Sect. 3.1 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 road segments. As a result, overall consumption may increase for a full battery; see Fig. 7a 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

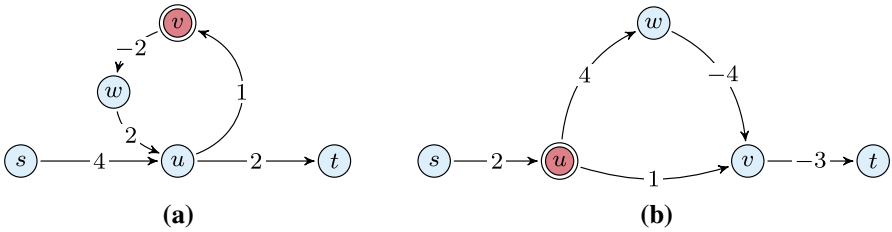

Fig. 7 Energy-optimal paths with charging stops. The battery capacity is $M = b_s = 5$. Charging stations are highlighted (red) and their SoC range is $[0, 5]$. **a** The energy-optimal s–t path $[s, u, v, w, u, t]$ contains a cycle, because a detour to the charging station v is necessary. Recharging 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 $r_v \in [1, 2]$ (Color figure online)

difficult compared to simpler models, which assume that charging always results in a full battery [61]. 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 Fig. 7a. One can even construct cases where an optimal path that requires no charging contains a subpath that can be improved via charging; see Fig. 7b. 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.

4.2 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 [49]. We now describe how multicriteria search 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 a multicriteria scenario, we can apply Pareto dominance to remove suboptimal labels; see Fig. 8a. 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.

Using the modified vertex labels, we outline an adaptation of the multicriteria shortest path algorithm [49]. 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 in 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 Fig. 8b. 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

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

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 Fig. 8c. 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 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. 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 Sect. 2, is conceptually simpler, can easily be integrated with known speedup techniques, and runs in polynomial time.

4.3 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

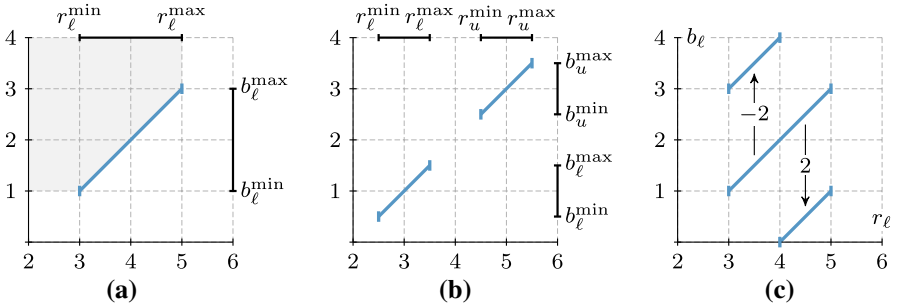

Fig. 8 Illustration of labels in the baseline approach. They map the amount of charged energy to the resulting SoC, assuming $M = 4$ (charging ranges can exceed the value 4 if multiple charging stops are required). **a** The blue segment shows different configurations of charging and resulting SoC of 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 a shift of the segment along the y-axis. Ranges shrink due to battery constraints, because certain subranges have insufficient charge or waste energy from recuperation (Color figure online)

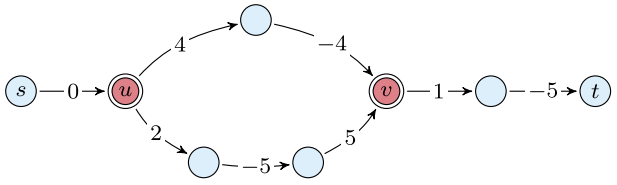

Fig. 9 Interdependence between initial SoC and the resulting optimal route with charging stops. Charging stations are highlighted (red) and have a charging range of $[0, 5]$, assuming a battery capacity of $M = 5$. Independent of the initial SoC $b_s \in [0, M]$ at the source s, the objective at v is 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 $b_s \in [4, 5]$ (Color figure online)

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 When reaching 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 $v \in S$. 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 9 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 [29] 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 1, there are at most $\mathcal{O}(|V|)$ 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* 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.

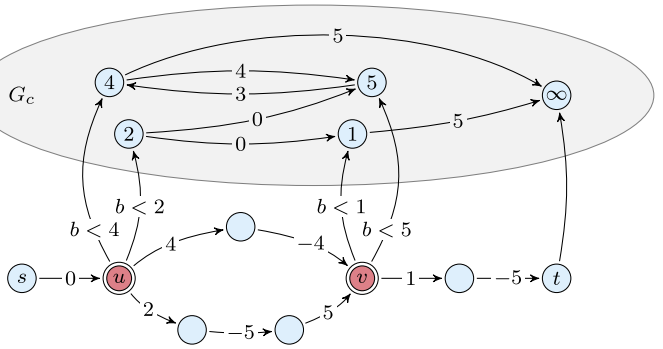

Fig. 10 Search graph for energy-optimal routes with charging stops, based on the original graph depicted in Fig. 9. Vertices in the charging station graph (shaded area) are labeled with their departure SoC. Edge labels indicate costs in the original graph, arrival SoC in the charging station graph, and SoC restrictions for transfer edges

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 Fig. 10 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 $b_P^{\text{dep}} \in [0, M]$ 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 $b^{\text{dep}}\colon V_c \to [0, M] \cup \{\infty\}$ and set $b^{\text{dep}}(u') := b_P^{\text{dep}}$. We also add a dummy target t' to V_c with $b^{\text{dep}}(t') := \infty$.

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 $b^{\text{dep}}(u') = b_P^{\text{dep}}$ to every charging vertex $v' \in V_c$ of v with $f_P(b_P^{\text{dep}}) < b^{\text{dep}}(v')$ to E_c. Together with the edge (u', v'), we also store the SoC upon arrival at v', i.e., we store a mapping $b^{\text{arr}}\colon E_c \to [0, M]$ and set $b^{\text{arr}}(u', v') := f_P(b_P^{\text{dep}})$.

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 $b^{\text{dep}}(v')$ 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

$$f_{(v,v')}(b) := \begin{cases} b & \text{if } b < b^{\text{dep}}(v'), \\ -\infty & \text{otherwise,} \end{cases} \tag{3}$$

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 $b^{\text{dep}}(v')$ may still be traversed in the original graph (without recharging at v) if $b \geq b^{\text{dep}}(v')$.

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 Sect. 3.1 to find energy-optimal routes in the presence of charging stations; see Algorithm 3 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 $v \in V \cup V_c$, 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 Algorithm 3. 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 Algorithm 3, by scanning transfer edges and making use of the artificial SoC functions according to Equation 3. Alternatively, if $u \in V_c$ is a charging station, it is handled separately by the algorithm; see Lines 16–22 in Algorithm 3. 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, predecessor pointers are used as in Dijkstra's algorihm. 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 Sect. 3.1 carry over to the charging station graph G_c, by setting the potential of every

Algorithm 3: EVD with charging stops.

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 $b_s \in [0, M]$.

Output: For each vertex $v \in V$, the maximum objective $b_v - r_v$ corresponding to the path that minimizes overall consumption.

```
// initialize labels
foreach v ∈ V ∪ V_c do
    ℓ(v) = (b_v, r_v) ⟵ (−∞, 0);
ℓ(s) ⟵ (b_s, 0);
Q.insert(s, b_s);
// run main loop
while Q.isNotEmpty() do
    u ⟵ Q.deleteMax();
    (b_u, r_u) ⟵ ℓ(u);
    if u ∈ V then
        // scan outgoing edges in the original graph and transfer edges
        foreach (u, v) ∈ E ∪ E_x do
            b ⟵ f_(u,v)(b_u);
            (b_v, r_v) ⟵ ℓ(v);
            if b − r_u > b_v − r_v then
                ℓ(v) ⟵ (b, r_u);
                Q.update(v, b − r_u);
    else
        // scan outgoing edges in the charging station graph
        foreach (u, v) ∈ E_c do
            b ⟵ b^arr(u, v);
            r ⟵ r_u + b^dep(u) − b_u;
            (b_v, r_v) ⟵ ℓ(v);
            if b − r > b_v − r_v then
                ℓ(v) ⟵ (b, r);
                Q.update(v, b − r);
```

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 can use the polynomial-time algorithm of Bellman–Ford [14,31] to compute a vertex-induced potential (see Sect. 3.1). 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 Sect. 3.2). Finally, the third step is solved by Algorithm 3. 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 3 summarizes the theoretical insights of this section.

Theorem 3 *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.*

4.4 A Heuristic Implementation

We discuss a practical variant of the EVD algorithm with charging stops introduced in Sect. 4.3. The construction of the subgraph G_c is time-consuming on realistic instances. Luckily, we can move most work to preprocessing, since paths between charging stations are independent of source and target. We also propose a simpler search graph, which can naturally be combined with CH for further speedup.

First, we obtain vertex potentials during preprocessing, using the more practical variants of EVD described in Sect. 3.1 instead of the algorithm of Bellman–Ford. 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 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 Algorithm 4 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 7 in Algorithm 4. If this yields an improvement to the label at v, it is updated

Algorithm 4: Heuristic variant of EVD with charging stops.

Input: 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.
Output: An SoC $b_v \in [0, M] \cup \{-\infty\}$ and a corresponding amount $r_v \in \mathbb{R}$ of charged energy for each $v \in V$.

```
1  initialize labels as in Algorithm 3;
2  while Q.isNotEmpty() do
3      u ⟵ Q.deleteMax();
4      (b_u, r_u) ⟵ ℓ(u);
5      if u ∈ S then
           // scan shortcuts between charging stations
6          foreach v ∈ S ∪ {t} do
7              b* ⟵ arg max_{b ∈ [max{b_u, b_u^min}, max{b_u, b_u^max}] ∪ {b_u}} f_(u,v)(b) − b;
8              r ⟵ r_u + b* − b_u;
9              b ⟵ f_(u,v)(b*);
10             (b_v, r_v) ⟵ ℓ(v);
11             if b − r > b_v − r_v then
12                 ℓ(v) ⟵ (b, r);
13                 Q.update(v, b − r);
14     else
15         scan outgoing edges in the original graph as in Algorithm 3;
```

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 Fig. 9 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 Sect. 6.3.

Integration with Contraction Hierarchies To enable faster queries, we propose CH, which have been extended to EV scenarios before [29]. In its basic variant, the speedup technique CH [33] iteratively *contracts* vertices in increasing order of (heuristic) importance during a preprocessing step, maintaining distances between all remaining vertices by adding *shortcut* edges, if necessary. *Witness searches* determine whether a shortcut is required to preserve distances. An online CH query runs bidirectional from source and target on the input graph augmented by all shortcuts added during preprocessing, following only *upward* edges (from less important to more important vertices).

Similar to previous approaches [59], 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 Sect. 3.2. 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 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 $c_e^{\max \le M} := \max_{b \in [0,M]} \{b - f_e(b) \mid f_e(b) \neq -\infty\}$ 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 $\bar{c} \in \mathbb{R}_{\ge 0}$ 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 [29].)

In summary, our preprocessing routine comprises three steps: (1) computation of a consistent potential function, (2) SoC 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, which contain original edges and shortcuts computed during preprocessing. In the component, the search scans only *upward* edges with respect to the vertex order. 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 Algorithm 3 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 [33]. 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 large 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 [37] in a natural way. The basic idea of A* search is to compute a consistent potential function on the vertices that changes the order in which vertices are extracted from the priority queue, such that vertices closer to the target are extracted first.

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. Sect. 3.1).

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. [33]. The rank of a vertex is then set to $4\,\mathrm{ED} + \mathrm{CQ}$ (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.

5 Extending Customizable Route Planning

Energy consumption is strongly influenced by a number of factors, such as vehicle load, auxiliary consumers, weather condition, driving style, and traffic conditions [65]. While some factors are static, others, such as weather conditions and vehicle load, are not. 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. [20], which exploits ideas from Multi-Level Dijkstra (MLD) [22,39,43,56,57]. 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 [20]. In this section, we tackle the problem setting introduced in Sect. 2.1. Hence, we do not consider stops at charging stations. In the following, we recap the preprocessing phase (Sect. 5.1) and the query phase (Sect. 5.2) of the MLD algorithm, describing our extensions along the way.

5.1 Preprocessing and Customization

The preprocessing phase computes a multilevel overlay [43,57] of the input graph $G = (V, E)$. It is obtained from a *nested multilevel partition* of the vertices of G, defined as follows. A *partition* of G is a family $\mathcal{V} = \{V_1, \ldots, V_k\}$ of *cells* $V_i \subseteq V$, 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, with $i \in \{1, \ldots, k\}$, 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 $\mathcal{V}^0 := \{\{v\} \mid v \in V\}$ and $\mathcal{V}^{L+1} := \{V\}$. An edge $(u, v) \in E$ is a *boundary edge* (u and v are *boundary vertices*) on level ℓ, if u and v are in different cells of $\mathcal{V}^\ell$. Note that a boundary vertex of level ℓ is also a boundary vertex of lower levels.

For a fixed level $\ell \in \{1, \ldots, L\}$, 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 [20]. 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

[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.

its SoC function. It proceeds in a bottom-up fashion, starting with the lowest level 1. Within a fixed level $\ell \in \{1, \ldots, L\}$, 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, like Delling et al. [20], when processing a level $\ell \in \{2, \ldots, L\}$, we make use of the already computed overlay 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 edge costs in plain CRP [20], 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. These logs are then merged sequentially after processing each level $\ell \in \{1, \ldots, L\}$.

Preliminary experiments indicated that more than 80% of the functions have simple form, so they can be compressed to constant size (see Sect. 3.2). Only for the remaining cases a thread uses its log, while compressed functions are written to the (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. [20], 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 Sect. 3.2).

Vertices of the input graph G are represented by indices $\{1, \ldots, |V|\}$ in our implementation. We 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 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 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* [11] 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* predecessor 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.

5.2 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 Sect. 3 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 [20]. 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).

Bidirectional Search We discuss techniques to accelerate SoC queries based on *bidirectional search*. Basically, bidirectional search simultaneously runs a *forward search* from s and a *backward search* from t 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 its backward graph 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 [24]. We denote them by *Bidirectional Profile-Evaluating MLD (BPE-MLD)* and *Bidirectional Distance-Bounding MLD (BDB-MLD)*. Both use a regular forward EVD search.

The first (straightforward) 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) at 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 Sect. 3.2 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. We stop the search as soon as the condition $k_F - k_B \leq \max\{b^*, 0\}$ holds.

Unfortunately, running a backward profile search can be costly. Therefore, the second (more sophisticated) 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 [36].) 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 using 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, 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 scanned by the backward search and $b(v) - \underline{c}(v) \leq \max\{b^*, 0\}$, or v was not scanned 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 query 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 $\{1, \ldots, |V|\}$ 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 predecessor vertices are always unique in this case (c. f. Sect. 5.1).

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 DFS from both the source and the target, until each encounter a boundary vertex.

6 Experiments

In what follows, we describe the experimental setup (Sect. 6.1). Afterwards, we evaluate our basic algorithms (Sect. 6.2), approaches for routes with charging stops (Sect. 6.3), and our customizable technique (Sect. 6.4). Finally, we also compare our new algorithms to previous approaches (Sect. 6.5).

6.1 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.

All reported query times are average values of 1000 queries. SoC queries assume a full battery $b_s = M$ at the source $s \in V$. Note that a lower initial SoC would only result in faster query times.

Our main benchmark instance, named Eur-PTV, is based on the road network of Western Europe, kindly provided by PTV AG.[2] We retrieved elevation information for the vertices from the freely available NASA Shuttle Radar Topography Mission (SRTM) dataset.[3] It covers large parts of the world with tiles at a resolution of three arc seconds (approximately 90 meters). The elevation of a vertex is obtained by bilinear interpolation from the four corners of the SRTM tile containing the vertex. 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).

Our energy consumption data originates from the Passenger Car and Heavy Duty Emission Model (PHEM), developed by the Graz University of Technology [38].

[2] http://www.ptvgroup.com.

[3] http://srtm.csi.cgiar.org.

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 on Emission Factors for Road Traffic (HBEFA) [38]. The HBEFA driving cycles cover a large variety of road categories, slopes, speed limits, and traffic situations. These cycles were evaluated using different EV configurations and vehicle types to generate energy consumption estimates for all available driving situations. 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 which 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. The resulting graph consists of 22 198 628 vertices and 51 088 095 edges.

We use two different EV models from PHEM. The first, denoted PG-16, is based on the real production vehicle Peugeot iOn and has a battery capacity of 16 kWh (corresponding to a range of 100–150 km). The second, EV-85, is based on an artificial PHEM model [66], for which we assume a larger battery capacity of 85 kWh (similar to that of current high-end Tesla models with a range of 400–500 km). To get the best possible cruising range, we disabled auxiliary consumers in both models. 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 an Peugeot iOn and the artificial model, respectively. Edge consumption values are stored in mWh, to avoid rounding issues along short-distance edges.

6.2 Basic Algorithms

We evaluate the variants of EVD and profile search discussed in Sect. 3. 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 Sect. 5.2. Instead, we generate *in-range* queries by first picking a source $s \in V$ uniformly at random, from which we run a preliminary search with initial SoC $b_s = M$. The target t is picked uniformly at random from its search space (i. e., all vertices within the vehicle's range).

Evaluating Queries Table 1 reports figures for our basic algorithms and 1000 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, 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 Sect. 3.1 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

Table 1 Evaluation of basic algorithms on Eur-PTV for both vehicle models

Algorithm	PG-16				EV-85			
	Custom.		Query		Custom.		Query	
	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

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

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 Fig. 11). 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 section. Finally, we see that—in contrast to time-dependent route planning [4,19,24]—profile queries 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 [3,52]. 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.

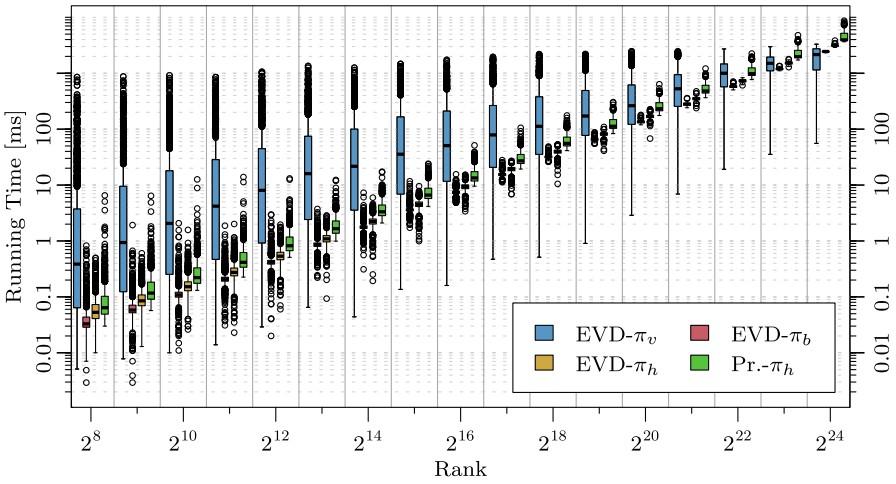

Fig. 11 Running times of basic algorithms subject to Dijkstra rank (EV-85). Low ranks indicate local queries. Battery capacity is increased to the point where range is not constrained

Hence, we use a slightly altered definition of Dijkstra rank: 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 $\{2^1, \ldots, 2^{\lfloor \log |V| \rfloor}\}$, we generate 1000 queries this way from sources chosen uniformly at random. To get meaningful results, we increase battery capacity from 85 to 1000 kWh, which corresponds to a cruising range of roughly 5000 km—enough to make the target reachable in all queries.

Figure 11 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 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 2 s. 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

Table 2 Impact of core size on performance (Ger-PTV, PG-16)

Core size		Prepr.	Query (ms)	
Ø Deg.	# Vertices	T. (s)	CH	CH + A*
0	–	176.8	526.3	1681.8
16	31,063 (0.66%)	257.5	16.3	16.9
32	5904 (0.13%)	416.7	12.0	4.9
48	3472 (0.07%)	548.5	11.2	4.2
64	2701 (0.06%)	633.7	11.8	4.1
128	2029 (0.04%)	786.8	12.6	6.4
∞	1966 (0.04%)	832.7	12.4	8.9

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

the lowest customization time (less than 0.5 s) 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 $\pi(v)$ on demand in the algorithm. This reduces space overhead to a *single* integer value (similar to EVD-sc.-g).

6.3 Routes with Charging Stops

To analyze our algorithms that allow intermediate stops at charging stations, we conduct experiments on Eur-PTV and a subnetwork, Ger-PTV, which represents Germany and consists of 4,692,091 vertices and 10,805,429 edges. Unless mentioned otherwise, we use 13,810 charging stations (1966 of them in Germany), which we located on ChargeMap.[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 1000 queries, with source and target vertices picked uniformly at random. All algorithms use height-induced potential functions.

Evaluating Queries Table 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, 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 2), this results in a speedup by a

[4] http://www.chargemap.com.

Table 3 Performance of approaches taking charging stops into account (Eur-PTV)

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
●	○	○	○	1487	759, 951	710.0	15,062	7,753,601	6285.7
●	●	○	○	2860	8433	309.6	3246	19,616	1281.5
●	●	●	○	2860	3563	128.2	3246	10,418	297.5
●	●	●	●	2860	1599	41.0	3246	9579	157.8

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

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 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 Sect. 4.2. It requires no preprocessing, but takes 20–30 s to answer queries, which is rather impractical. Simply plugging in the charging station graph G_S and using the modified EVD (see Sect. 4.4) 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% 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), as more charging stations are reachable from each station.

Table 4 Performance for varying distributions of charging stations (Ger-PTV, PG-16)

Scenario	$\|S\|$	Prepr.		Queries		
		T. (s)	$\|E_S\|$	#V. Sc.	#E. Sc.	T. (ms)
reg-cm	1966	548.5	539,145	4592	125,535	4.22
mix-cm	1966	548.1	539,145	4592	125,381	4.19
reg-r0.01	469	487.2	22,231	2234	50,070	1.30
reg-r0.1	4692	582.7	2,263,310	8904	223,779	7.97
reg-r1.0	46, 920	965.0	227,514,459	60,527	1,828,581	73.46

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 average vertex scans (#V. Sc.) and edge scans (#E. Sc.)

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, 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 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 in fact higher than the current number of gas stations in Germany).

Figure 12 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.

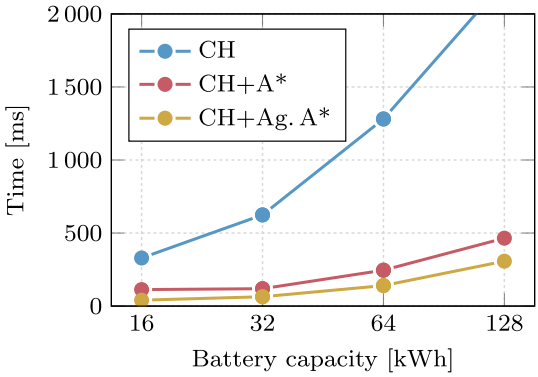

Fig. 12 Running times subject to cruising range (Eur-PTV, PG-16). Each point in the plot corresponds to the median running time of 1000 queries for one of the different approaches (CH, CH with A*, CH with aggressive A*) and varying battery capacities

6.4 Customizable Energy-Optimal Routes

Since our implementation of MLD exploits parallelism in both metric-dependent preprocessing and queries, experiments reported in this section were conducted on machine-p. As partitioning tool we used PUNCH (Partitioning Using Natural Cut Heuristics) [21], 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 5 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 1000 queries as in Sect. 6.2. 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, customization takes less than 4 s when parallelized, enabling frequent metric updates for the whole network. Executed sequentially, customization takes 34.8 s (respective 40.4 s) 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. 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, which we attribute to its larger number of negative edges.

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

Table 5 Evaluation of MLD approaches for both vehicle models (Eur-PTV)

Algorithm	PG-16				EV-85			
	Custom.		Query		Custom.		Query	
	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	2305	1.24
BPE-MLD	13.6	3.20	891	0.30	14.5	3.67	2194	0.92
BDB-MLD	13.6	3.20	1120	0.25	14.5	3.67	2754	0.67
Pr.-MLD	13.6	3.20	1068	0.75	14.5	3.67	2763	3.60

We report figures as in Table 1, for the same set of 1000 queries

2000 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 [20]: 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 Sect. 6.2, 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.

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 13 shows scalability of our MLD algorithms, using the Dijkstra rank method as explained in Sect. 6.2. For the same set of 1000 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 1000 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 maximal 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 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 likely consist of more breakpoints and shortcut scans become particularly expensive. As a result, we obtain maximal profile

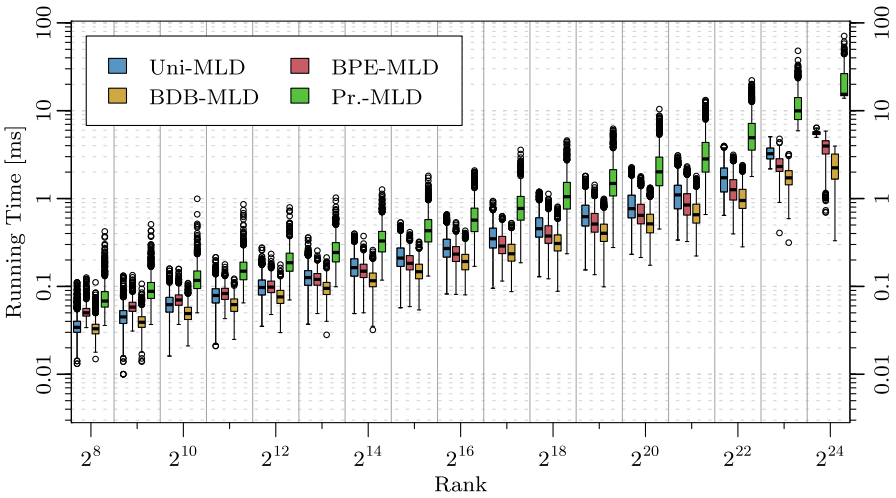

Fig. 13 Running times of MLD subject to Dijkstra rank. We use the same vehicle model and queries as in Fig. 11. Battery capacity is increased such that range is not constrained

query times of over 70 ms. Still, MLD yields a speedup of more than two orders of magnitude compared to plain profile search.

Figure 14 shows running times subject to Dijkstra rank for the PG-16 model. Again, we use the same set of queries as in Fig. 11. However, in contrast to Fig. 13, we 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 Fig. 13, 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.

6.5 Comparison of Approaches

We compare the performance of different approaches that compute routes for EVs (without charging stops). First, we consider the fastest previous 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 The fastest available approach that computes energy-optimal routes for EVs is based on CH [29,60]. The authors adapt plain CH [33] to the scenario of optimizing energy consumption in the following way: To avoid

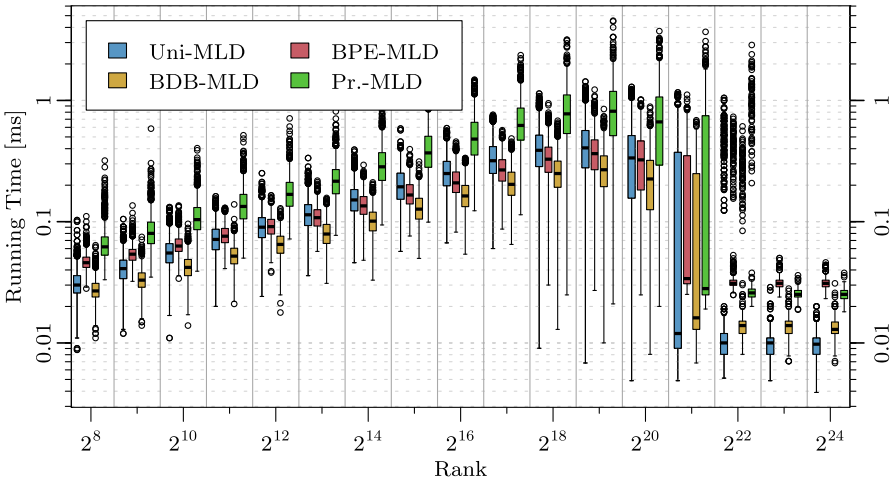

Fig. 14 Running times of the PG-16 model subject to Dijkstra rank, with reachability flags enabled. As in Fig. 11, smaller ranks indicate more local queries

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 [60], which was kindly given to us by the authors. The instance is based on an OpenStreetMap (OSM)[5] export of the road network of Japan, augmented with SRTM data. It has 25,970,678 vertices and 54,141,580 edges. 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. Using PUNCH [21], computing the partition took less than half an hour.

Eisner et al. [29,60] use a simple consumption model, where energy consumption of an edge $e = (u, v) \in E$ is assumed to depend only on horizontal distance $d(e)$ of the edge and vertical heights $h(u)$ and $h(v)$ of the vertices. More precisely, the energy consumption $c(e)$ of the edge e is

$$c(e) := \begin{cases} \kappa \cdot d(e) + \lambda \cdot (h(v) - h(u)) & \text{if } h(v) - h(u) \geq 0, \\ \kappa \cdot d(e) + \mu \cdot (h(v) - h(u)) & \text{otherwise.} \end{cases} \tag{4}$$

In accordance with the previous studies [60], we set $\kappa = 0.02$, $\lambda = 1$, and $\mu = 0.25$. Note that when applying this model to the European network, we observe that the amount of negative edges drops to 4.4%. Further, note that a height-induced potential

[5] http://www.openstreetmap.org.

Table 6 Comparison of speedup techniques for SoC queries (Jap-OSM)

Algorithm	Custom.		Query	
	Space (B/n)	Time (s)	# Vertex Scans	Time (ms)
EVD	4.0	–	12,661,423	2044.63
EVD [60]	4.0	–	14,431,809	6492.58
CH [60]	23.0	14,329.87	10,024	44.93
CH (scal.) [60]	23.0	7188.77	10,024	14.15
CH [our]	44.8	1076.74	252	0.88
Uni-MLD	7.7	1.83	2196	0.67
BPE-MLD	7.7	1.83	2252	0.62
BDB-MLD	7.7	1.83	2650	0.46

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 [60], we also report scaled timings

follows from the model (we set $\alpha := -\mu$). Therefore, computing the potential function does not require any customization time. Similar to Eisner et al. [29,60], we assume a very large battery capacity. As a result, the target is always in range and we disable reachability flags.

Table 6 reports results on Jap-OSM for our implementation of CH following the description in Sect. 4.4, 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 [60]. 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. [29,60] 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 approaches. For query times of CH, we obtain a scaling factor based on the EVD implementations, maintaining their speedup 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 uses a stopping criterion: The reported search space is about 56% of the graph size [29,60].

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 [29,60]. The latter can be explained by the fact that, 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, contract-

Table 7 Comparison of energy-optimal routes to other metrics

Instance	Travel time			Distance		
	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%

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

ing 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 s and is more than 50 times as fast as CH preprocessing. Our findings add to previous observations that, compared to CH [20], separator-based approaches are more robust towards metrics other than (unconstrained) travel time [11,20,25]. 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. [55] (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 7 shows results for Eur-PTV and Jap-OSM. We use the same 1000 queries as in Table 1 for Eur-PTV and Table 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 when using 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.

7 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 model 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 [29,51] and $\mathcal{NP}$-hard time-constrained variants that include charging stops [8,62] (which generalize the problem setting considered in this work). 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. 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 [20], 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 route planning for 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. Realistic models 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 include reducing the number of edges in the overlay of charging stations for better performance and scalability of CH [20,39,57] or integration with Customizable Contraction Hierarchies (CCH) [25] for faster preprocessing. It might also be worthwhile to extend the proposed A* search to an adaptation of ALT [34] for faster queries. Moreover, one could consider a *profile* variant of this problem setting, i. e., ask for an SoC profile with intermediate charging stops. The problem setting could also be extended to account for time spent at charging stations in optimization, e. g., by restricting the number of charging stops [61]. Note that adding travel time (including charging time) as an optimization criterion typically results in $\mathcal{NP}$-hard problem settings [8,62].

Finally, note that 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 1 in Sect. 2.2). Then, the parameter M could be part of the query input.

Acknowledgements We would like to thank Raphael Luz for providing the consumption data [38,66], Renato Werneck for running PUNCH [21], Moritz Kobitzsch for interesting discussions, and Christian Schulz and Dennis Luxen for providing Buffoon [53] and OSRM [48], respectively, which we used in our preliminary experiments. We thank Sabine Storandt for making Jap-OSM available, and Konstantinos Demestichas for providing sample data on energy consumption of EVs.

Funding Funding was provided by Deutsche Forschungsgemeinschaft (Grant No. WA 654/23-1).

References

1. Artmeier, A., Haselmayr, J., Leucker, M., Sachenbacher, M.: 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), Lecture Notes in Computer Science, vol. 6359, pp. 309–316. Springer (2010)
2. Atallah, M.J.: Some dynamic computational geometry problems. Comput. Math. Appl. **11**(12), 1171–1181 (1985)
3. Bast, H., Delling, D., Goldberg, A.V., Müller-Hannemann, M., Pajor, T., Sanders, P., Wagner, D., Werneck, R.F.: Route Planning in Transportation Networks, Lecture Notes in Computer Science, vol. 9220, pp. 19–80. Springer (2016)
4. Batz, G.V., Geisberger, R., Sanders, P., Vetter, C.: Minimum time-dependent travel times with contraction hierarchies. ACM J. Exp. Algorithmics **18**, 1.4:1–1.4:43 (2013)
5. Batz, G.V., Sanders, P.: Time-dependent route planning with generalized objective functions. In: Proceedings of the 20th Annual European Symposium on Algorithms (ESA'12), Lecture Notes in Computer Science, vol. 7501, pp. 169–180. Springer (2012)
6. Bauer, R., Delling, D., Sanders, P., Schieferdecker, D., Schultes, D., Wagner, D.: Combining hierarchical and goal-directed speed-up techniques for Dijkstra's algorithm. ACM J. Exp. Algorithmics **15**, 2.3:1–2.3:31 (2010)
7. Baum, M.: Engineering Route Planning Algorithms for Battery Electric Vehicles. Phd thesis, Karlsruhe Institute of Technology (2018)
8. Baum, M., Dibbelt, J., Gemsa, A., Wagner, D., Zündorf, T.: 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), pp. 44:1–44:10. ACM (2015)
9. Baum, M., Dibbelt, J., Hübschle-Schneider, L., Pajor, T., Wagner, D.: 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), OpenAccess Series in Informatics (OASIcs), vol. 42, pp. 138–151. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2014)
10. Baum, M., Dibbelt, J., Pajor, T., Wagner, D.: Energy-optimal routes for electric vehicles. In: Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'13), pp. 54–63. ACM (2013)
11. Baum, M., Dibbelt, J., Pajor, T., Wagner, D.: Dynamic time-dependent route planning in road networks with user preferences. In: Proceedings of the 15th International Symposium on Experimental Algorithms (SEA'16), Lecture Notes in Computer Science, vol. 9685, pp. 33–49. Springer (2016)
12. Baum, M., Dibbelt, J., Wagner, D., Zündorf, T.: Modeling and engineering constrained shortest path algorithms for battery electric vehicles. In: Proceedings of the 25th Annual European Symposium on Algorithms (ESA'17), Leibniz International Proceedings in Informatics (LIPIcs), vol. 87, pp. 11:1–11:16. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2017)

13. Baum, M., Sauer, J., Wagner, D., Zündorf, T.: Consumption profiles in route planning for electric vehicles: theory and applications. In: Proceedings of the 16th International Symposium on Experimental Algorithms (SEA'17), Leibniz International Proceedings in Informatics (LIPIcs), vol. 75, pp. 19:1–19:18. Schloss Dagstuhl–Leibniz-Zentrum für Informatik (2017)
14. Bellman, R.: On a routing problem. Q Appl. Math. **16**(1), 87–90 (1958)
15. Cherkassky, B.V., Georgiadis, L., Goldberg, A.V., Tarjan, R.E., Werneck, R.F.: Shortest-path feasibility algorithms: an experimental evaluation. ACM J. Exp. Algorithmics **14**, 2.7:1–2.7:37 (2010)
16. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)
17. Davenport, H., Schinzel, A.: A combinatorial problem connected with differential equations. Am. J. Math. **87**(3), 684–694 (1965)
18. Dean, B.C.: Shortest Paths in FIFO Time-Dependent Networks: Theory and Algorithms. Technical Report, Massachusetts Institute of Technology (2004)
19. Delling, D.: Time-dependent SHARC-routing. Algorithmica **60**(1), 60–94 (2011)
20. Delling, D., Goldberg, A.V., Pajor, T., Werneck, R.F.: Customizable route planning in road networks. Transp. Sci. **51**(2), 566–591 (2017)
21. Delling, D., Goldberg, A.V., Razenshteyn, I., Werneck, R.F.: Graph partitioning with natural cuts. In: Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS'11), pp. 1135–1146. IEEE (2011)
22. Delling, D., Holzer, M., Müller, K., Schulz, F., Wagner, D.: High-performance multi-level routing, dimacs series. In: Discrete Mathematics and Theoretical Computer Science, vol. 74, pp. 73–92. American Mathematical Society (2009)
23. Delling, D., Wagner, D.: Landmark-based routing in dynamic graphs. In: Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07), Lecture Notes in Computer Science, vol. 4525, pp. 52–65. Springer (2007)
24. Delling, D., Wagner, D.: Time-Dependent Route Planning, Lecture Notes in Computer Science, vol. 5868, pp. 207–230. Springer (2009)
25. Dibbelt, J., Strasser, B., Wagner, D.: Customizable contraction hierarchies. ACM J. Exp. Algorithmics **21**, 1.5:1–1.5:49 (2016)
26. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numer. Math. **1**(1), 269–271 (1959)
27. Dreyfus, S.E.: An appraisal of some shortest-path algorithms. Oper. Res. **17**(3), 395–412 (1969)
28. Efentakis, A., Pfoser, D.: Optimizing landmark-based routing and preprocessing. In: Proceedings of the 6th ACM SIGSPATIAL International Workshop on Computational Transportation Science (IWCTS'13), pp. 25–30. ACM (2013)
29. Eisner, J., Funke, S., Storandt, S.: Optimal route planning for electric vehicles in large networks. In: Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI'11), pp. 1108–1113. AAAI Press (2011)
30. Fiori, C., Ahn, K., Rakha, H.A.: Power-based electric vehicle energy consumption model: model development and validation. Appl. Energy **168**, 257–268 (2016)
31. Ford, L.R.: Network Flow Theory. Technical Report P-923, Rand Corporation, Santa Monica, California (1956)
32. Foschini, L., Hershberger, J., Suri, S.: On the complexity of time-dependent shortest paths. Algorithmica **68**(4), 1075–1097 (2014)
33. Geisberger, R., Sanders, P., Schultes, D., Vetter, C.: Exact routing in large road networks using contraction hierarchies. Transp. Sci. **46**(3), 388–404 (2012)
34. Goldberg, A.V., Harrelson, C.: Computing the shortest path: a* search meets graph theory. In: Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05), pp. 156–165. SIAM (2005)
35. Goodrich, M.T., Pszona, P.: 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), pp. 193–202. ACM (2014)
36. Gutman, R.J.: 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), pp. 100–111. SIAM (2004)
37. Hart, P.E., Nilsson, N.J., Raphael, B.: A formal basis for the heuristic determination of minimum cost paths. IEEE Trans. Syst. Sci. Cybern. **4**(2), 100–107 (1968)

38. Hausberger, S., Rexeis, M., Zallinger, M., Luz, R.: Emission Factors from the Model PHEM for the HBEFA Version 3. Technical Report I-20/2009, University of Technology, Graz (2009)
39. Holzer, M., Schulz, F., Wagner, D.: Engineering multilevel overlay graphs for shortest-path queries. ACM J. Exp. Algorithmics **13**, 2.5:1–2.5:26 (2009)
40. Huber, G., Bogenberger, K.: Long-trip optimization of charging strategies for battery electric vehicles. Transp. Res. Record: J. Transp. Res. Board **2497**, 45–53 (2015)
41. Johnson, D.B.: A note on Dijkstra's shortest path algorithm. J. ACM **20**(3), 385–388 (1973)
42. Johnson, D.B.: Efficient algorithms for shortest paths in sparse networks. J. ACM **24**(1), 1–13 (1977)
43. Jung, S., Pramanik, S.: An efficient path computation model for hierarchically structured topographical road maps. IEEE Trans. Knowl. Data Eng. **14**(5), 1029–1046 (2002)
44. Kluge, S., Sánta, C., Dangl, S., Wild, S.M., Brokate, M., Reif, K., Busch, F.: On the computation of the energy-optimal route dependent on the traffic load in Ingolstadt. Transp. Res. Part C: Emerg. Technol. **36**, 97–115 (2013)
45. Kobayashi, Y., Kiyama, N., Aoshima, H., Kashiyama, M.: 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), pp. 920–925. IEEE (2011)
46. Liao, C.S., Lu, S.H., Shen, Z.J.M.: The electric vehicle touring problem. Transp. Res. Part B: Methodol. **86**, 163–180 (2016)
47. Liu, C., Wu, J., Long, C.: Joint charging and routing optimization for electric vehicle navigation systems. In: Proceedings of the 19th International Federation of Automatic Control World Congress (IFAC'14), IFAC Proceedings Volumes, vol. 47, pp. 9611–9616. Elsevier (2014)
48. Luxen, D., Vetter, C.: Real-time routing with OpenStreetMap data. In: Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS'11), pp. 513–516. ACM (2011)
49. Martins, E.Q.V.: On a multicriteria shortest path problem. Eur. J. Oper. Res. **16**(2), 236–245 (1984)
50. Orda, A., Rom, R.: Minimum weight paths in time-dependent networks. Networks **21**(3), 295–319 (1991)
51. Sachenbacher, M., Leucker, M., Artmeier, A., Haselmayr, J.: Efficient energy-optimal routing for electric vehicles. In: Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI'11), pp. 1402–1407. AAAI Press (2011)
52. Sanders, P., Schultes, D.: Highway hierarchies hasten exact shortest path queries. In: Proceedings of the 13th Annual European Conference on Algorithms (ESA'05), Lecture Notes in Computer Science, vol. 3669, pp. 568–579. Springer (2005)
53. Sanders, P., Schulz, C.: Distributed evolutionary graph partitioning. In: Proceedings of the 14th Meeting on Algorithm Engineering & Experiments (ALENEX'12), pp. 16–29. SIAM (2012)
54. Schönfelder, R., Leucker, M.: Abstract routing models and abstractions in the context of vehicle routing. In: Proceedings of the 24th International Joint Conference on Artificial Intelligence (IJCAI'15), pp. 2639–2645. AAAI Press (2015)
55. Schönfelder, R., Leucker, M., Walther, S.: Efficient profile routing for electric vehicles. In: Proceedings of the 1st International Conference on Internet of Vehicles (IOV'14), Lecture Notes in Computer Science, vol. 8662, pp. 21–30. Springer (2014)
56. Schulz, F., Wagner, D., Weihe, K.: Dijkstra's algorithm on-line: an empirical case study from public railroad transport. ACM J. Exp. Algorithmics **5**, 12:1–12:23 (2000)
57. Schulz, F., Wagner, D., Zaroliagis, C.: Using multi-level graphs for timetable information in railway systems. In: Proceedings of the 4th Workshop on Algorithm Engineering & Experiments (ALENEX'02), Lecture Notes in Computer Science, vol. 2409, pp. 43–59. Springer (2002)
58. Smith, O.J., Boland, N., Waterer, H.: Solving shortest path problems with a weight constraint and replenishment arcs. Comput. Oper. Res. **39**(5), 964–984 (2012)
59. Storandt, S.: 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), pp. 20–25. ACM (2012)
60. Storandt, S.: Algorithms for Vehicle Navigation. Ph.D. thesis, Universität Stuttgart (2013)
61. Storandt, S., Funke, S.: Cruising with a battery-powered vehicle and not getting stranded. In: Proceedings of the 26th AAAI Conference on Artificial Intelligence (AAAI'12), pp. 1628–1634. AAAI Press (2012)
62. Strehler, M., Merting, S., Schwan, C.: Energy-efficient shortest routes for electric and hybrid vehicles. Transp. Res. Part B: Methodol. **103**, 111–135 (2017)

63. Sun, Z., Zhou, X.: To save money or to save time: intelligent routing design for plug-in hybrid electric vehicle. Transp. Res. Part D: Transp. Environ. **43**, 238–250 (2016)
64. Sweda, T.M., Dolinskaya, I.S., Klabjan, D.: Adaptive Routing and Recharging Policies for Electric Vehicles. Working paper no. 14-02, Northwestern University, Illinois (2014)
65. Sweeting, W.J., Hutchinson, A.R., Savage, S.D.: Factors affecting electric vehicle energy consumption. Int. J. Sustain. Eng. **4**(3), 192–201 (2011)
66. Tielert, T., Rieger, D., Hartenstein, H., Luz, R., Hausberger, S.: Can V2X communication help electric vehicles save energy? In: Proceedings of the 12th International Conference on ITS Telecommunications (ITST'12), pp. 232–237. IEEE (2012)
67. Wang, Y., Jiang, J., Mu, T.: Context-aware and energy-driven route optimization for fully electric vehicles via crowdsourcing. IEEE Trans. Intell. Transp. Syst. **14**(3), 1331–1345 (2013)
68. Wiernik, A., Sharir, M.: Planar realizations of nonlinear Davenport–Schinzel sequences by segments. Discret. Comput. Geom. **3**(1), 15–47 (1988)

Affiliations

Moritz Baum[1] · Julian Dibbelt[2] · Thomas Pajor[2] · Jonas Sauer[1] · Dorothea Wagner[1] · Tobias Zündorf[1]

✉ Moritz Baum
moritz.baum@kit.edu

Jonas Sauer
jonas.sauer2@kit.edu

Dorothea Wagner
dorothea.wagner@kit.edu

Tobias Zündorf
tobias.zuendorf@kit.edu

[1] Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany

[2] Sunnyvale, USA