Energy-Optimal Routes for Electric Vehicles*

Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner

Karlsruhe Institute of Technology (KIT), 76128 Karlsruhe, Germany.
{moritz.baum,dibbelt,pajor,dorothea.wagner}@kit.edu

Abstract. We study the problem of electric vehicle route planning, where an important aspect is computing paths that minimize energy consumption. Thereby, any method must cope with specific properties, such as recuperation, battery constraints (over- and under-charging), and frequently changing cost functions (e. g., due to weather conditions). This work presents a practical algorithm that quickly computes energy-optimal routes for networks of continental scale. Exploiting multi-level overlay graphs [26, 31], we extend the Customizable Route Planning approach [8] to our scenario in a sound manner. This includes the efficient computation of profile queries and the adaption of bidirectional search to battery constraints. Our experimental study uses detailed consumption data measured from a production vehicle (Peugeot iOn). It reveals for the network of Europe that a new cost function can be incorporated in about five seconds, after which we answer random queries within 0.3 ms on average. Additional evaluation on an artificial but realistic [22, 36] vehicle model with unlimited range demonstrates the excellent scalability of our algorithm: Even for long-range queries across Europe it achieves query times below 5 ms on average—fast enough for interactive applications. Altogether, our algorithm exhibits faster query times than previous approaches, while improving (metric-dependent) preprocessing time by three orders of magnitude.

1 Introduction

Web-based route planning and on-board navigation have become a commodity for millions of users. A particularly important problem is the explicit consideration of electric vehicles that usually employ a rather limited cruising range. To overcome *range anxiety*—the fear of getting stranded—careful guidance of the user is crucial. We, therefore, study the problem of quickly computing routes that minimize *energy consumption* in order to maximize cruising range. This imposes two nontrivial challenges: *Recuperation* allows the vehicle to recharge its battery (e. g., when driving downhill), and the *battery capacity* imposes constraints on the range and the amount of recuperable energy. Modeling energy consumption precisely is thereby important, since it is strongly influenced by factors, such as the vehicle's load, auxiliary consumers, weather condition, driving style, and traffic conditions [35]. While some factors are static, others, such as weather conditions and vehicle load, are not.

Related Work Route planning on road networks in general has seen substantial algorithmic progress over the past years. While the problem can be solved by Dijkstra's algorithm [16], it is too slow in practice. Therefore, a multitude of *speedup techniques* use an offline preprocessing phase to accelerate queries (see [11, 32] for surveys). Most were developed for static arc costs representing travel times. A notable exception is Customizable Route Planning (CRP) by Delling et al. [8]. Based on multilevel overlay graphs [10, 24, 26, 30, 31], it is explicitly designed to work with arbitrary metrics and is able to integrate new cost functions in mere seconds. Regarding electric vehicles, Artmeier at al. [1] observe that using energy consumption as metric may result in negative cost values for some arcs (though, physical constraints prohibit negative cycles). They use a simplistic energy consumption model and handle negative costs by variants of

* Support by DFG grant WA 654/16-1, by the EU FP7/2007-2013 (DG CONNECT.H5 – Smart Cities and Sustainability), under grant agreement no. 288094 (project eCOMPASS), and by the Federal Ministry of Economics and Technology under grant no. 01ME12013 (project iZeus).

label-*correcting* Dijkstra [15] and Bellman-Ford [4] algorithms, from which the latter turns out to be too slow in practice. To get rid of negative arc costs, one can use a technique called *potential shifting* [25]. This re-enables Dijkstra's algorithm. Using a (slightly) more realistic physical energy consumption (from which potentials are directly obtained) [28], Artmeier et al. combine potential shifting with goal-directed search [21] to get a factor of three speedup over their (previous) label-correcting approach. Demestichas et al. [14] use machine learning to obtain consumption data from real electric vehicles in field tests. However, their work does not focus on routing algorithms. In contrast, Eisner et al. [18, 34] use a relatively simple consumption model based on distance and height difference. For route planning they go beyond potential shifting, and integrate battery constraints into the arc costs by modeling them as piecewise linear functions. This is similar to the time-dependent case [3, 12, 17], however, function complexity is much lower. Exploiting this similarity, the authors adapt Contraction Hierarchies (CH) [3, 19] to the scenario of optimizing energy consumption. To handle functional arc costs in witness searches [19] during preprocessing, they do not perform profile queries [7, 12]. Instead, they acquire upper bounds on witness paths by sampling, in order to improve preprocessing time at the cost of query speed. Using this approach, query times of 38–45 ms are achieved on country-scale networks after several hours of preprocessing [34].

Our Contribution In this work, we present a practical approach to optimize energy consumption for electric vehicles that is fast both in (metric-dependent) preprocessing as well as in query speed, even on continental networks. For that, we carefully extend the Customizable Route Planning (CRP) method of Delling et al. [8] to handle recuperation (i. e., negative costs) and battery capacity constraints. Modeling such constraints as a special form of piecewise linear functions enables us to compute *profile queries* efficiently—a crucial ingredient for CRP's preprocessing in our scenario. Thereby, we achieve fast (metric-dependent) preprocessing of the whole network within a few seconds. This enables flexible updates, e.g., due to hourly weather forecasts, or refinements of the underlying consumption model (as is necessary when machine learning approaches are used to improve the model [14]). Moreover, we present several query algorithms, from which the most sophisticated variant simultaneously, and in parallel, employs a backward search (from the destination t) that helps bound the forward search in order to "guide" it toward t (similarly to [12, 20]), while considering battery constraints. In contrast to previous work, our experimental study is based on highly detailed consumption data measured from a real production vehicle (Peugeot iOn). It turns out that this vehicle model is harder for (our) algorithms than previously considered (simpler) consumption models. On the continental road network of Europe, we show that using our approach, energy-optimal routes can be computed in 1.1 ms time, easily enabling interactive applications. Moreover, we show that our algorithms scale excellently with the available cruising range, making them robust to future developments in increasing battery capacities.

This paper is organized as follows. Section 2 sets necessary notion. Section 3 introduces our approach to electric vehicle route planning and describes how we compute profile queries. Section 4 uses these ingredients to carefully extend CRP. Section 5 contains our experimental study, while Section 6 concludes with interesting open problems.

2 Preliminaries

We model the road network as a (directed) *graph* $G = (V, A)$ with $n = |V|$ *vertices* and $m = |A|$ *arcs*. Intersections are represented by vertices $v \in V$, road segments by arcs $(u, v) \in A$. Vertices have associated height values in meters (which are relevant to model energy consumption) given by a function $h\colon V \to \mathbb{N}$. Arcs have associated distance values in meters (again, relevant for

consumption) given by a function $\mathrm{dist}\colon A \to \mathbb{N}$. An *s–t path* P (in G), sometimes written $P_{s,t}$, is a sequence $P = [v_1 = s, v_2, \ldots, v_k = t]$ of vertices such that $(v_i, v_{i+1}) \in A$. If s and t coincide, we call P a *cycle*.

To model energy consumption, we assign each arc $a = (u, v)$ a *cost function* f_a that maps *battery state of charge (SoC) levels* $b(u)$ (at vertex u; measured in mWh) to an energy consumption value (also measured in mWh) which is required to traverse the arc a. The function f_a takes battery constraints into account [18], i. e., the *maximum charge level*—or capacity—of the battery M, as well as the minimum SoC required for traversing a. See Fig. 1b-1c for an example. We, therefore, consider a *function space* $\mathbb{F}$ consisting of functions of the form $f\colon \mathbb{R} \cup \{-\infty\} \to \mathbb{R} \cup \{\infty\}$, where ∞ and $-\infty$ are special values to handle insufficient charge. Our functions (must) fulfill the *FIFO property*, that is, for any SoC values $x \leq y$ it must hold that $x - f(x) \leq y - f(y)$. In other words, having lower SoC values never improves energy consumption. On $\mathbb{F}$ we require binary *link* (composition) and *merge* operations. Given two functions $f, g \in \mathbb{F}$, the link operation is defined as $\mathrm{link}(f, g) := f + g \circ (\mathrm{id} - f)$, whereas we define merging f and g by $\mathrm{merge}(f, g) := \min(f, g)$. Note that $\mathbb{F}$ is closed under both link and merge. The cost function f_P of a path P is defined by iteratively applying the link function. The cost of an s–t path P with initial SoC $b(s) = b_s$ is $f_P(b(s))$. Finally, an s–t path P with initial SoC b_s is called *optimal* (wrt. energy consumption), iff $f_P(b_s)$ is minimal among all s–t paths. Note that, by construction, infeasible paths, i. e., paths with $b(v_i) < 0$ for any i, have infinite cost. By $\underline{f}$, we denote the *lower bound* of f, and by $\overline{f}$ its *finite upper bound*, i. e., the largest value of f that is smaller than infinity (if such a value exist). Finally, we require designated functions $f^0(\cdot) = 0$ and $f^\infty(\cdot) = \infty$.

Fig. 1 illustrates our cost functions by using a small example graph. It shows the resulting functions after applying link and merge operations. Note that cost functions $f \in \mathbb{F}$ are piecewise linear, but not necessarily continuous, with varying degree of complexity: For *single* arcs $a = (u, v)$, functions f_a have a *simple form*: Let c_a denote the constant (i. e., ignoring battery constraints) *consumption cost* of arc a. Then, for a positive arc, i. e., for which c_a is positive, the arc's cost function is $f_a(b) = \infty$ for all SoC values $b < c_a$, and $f_a(b) = c_a$, otherwise. For a negative arc a, i. e., for which $c_a < 0$, we set $f_a(b) = \max(c_a, b - M)$. Note that $\underline{f}_a = c_a$. When linking a single path P of negative and positive arcs, the resulting function f_P is, unlike in time-dependent route planning [11], of very limited complexity [18]. However, merging different paths might also result in functions of $\mathcal{O}(|A|)$ line segments [6] (c. f. Fig. 1).

A *potential* is a function $\Pi\colon V \to \mathbb{R}$ on the vertices. If for every arc $a = (u, v)$ the condition $\underline{f}_a + \Pi(u) - \Pi(v) \geq 0$ holds, we call the potential *feasible*. Any (feasible) potential induces a graph G' of non-negative *reduced arc costs* by *shifting* the cost function at every arc $a = (u, v)$, setting $f'_a = f_a + \Pi(u) - \Pi(v)$.

A *partition* of the vertices V is a family $\mathcal{C} = \{C_1, \ldots, C_k\}$ of *cells* $C_i \subseteq V$, such that each vertex $v \in V$ is contained in exactly one cell C_i. More generally, a *nested multilevel partition* of L levels is a family $\{\mathcal{C}^1, \ldots, \mathcal{C}^L\}$ of partitions with nested cells, that is, for each level $\ell \leq L$ and cell $C_i^\ell \in \mathcal{C}^\ell$ there must exist a cell $C_j^{\ell+1} \in \mathcal{C}^{\ell+1}$ on level $\ell + 1$, such that $C_i^\ell \subseteq C_j^{\ell+1}$ holds. We call $C_j^{\ell+1}$ the *supercell* of C_i^ℓ. For consistency, we define $\mathcal{C}^0 = V$ and $\mathcal{C}^{L+1} = \{V\}$. An arc $(u, v) \in A$ is called a *boundary arc* on level ℓ, iff u and v are in different cells of $\mathcal{C}^\ell$. In this case, u and v are called *boundary vertices* (of level ℓ). Note that a boundary vertex of level ℓ is also a boundary vertex on all lower levels. Many general graph partitioning algorithms are available, several of which aim for balanced cells while minimizing the number of boundary arcs [2]. For road networks, tailored algorithms, such as PUNCH [9] and BUFFOON [29], exist.

In this work, we study two *problems: SoC queries* and *profile queries*. In an SoC query, one is given source and target vertices $s, t \in V$, and an initial SoC b_s. It asks for a (single) optimal s–t path departing at s with SoC b_s. A profile query does not take b_s as input, but asks for a *set* of

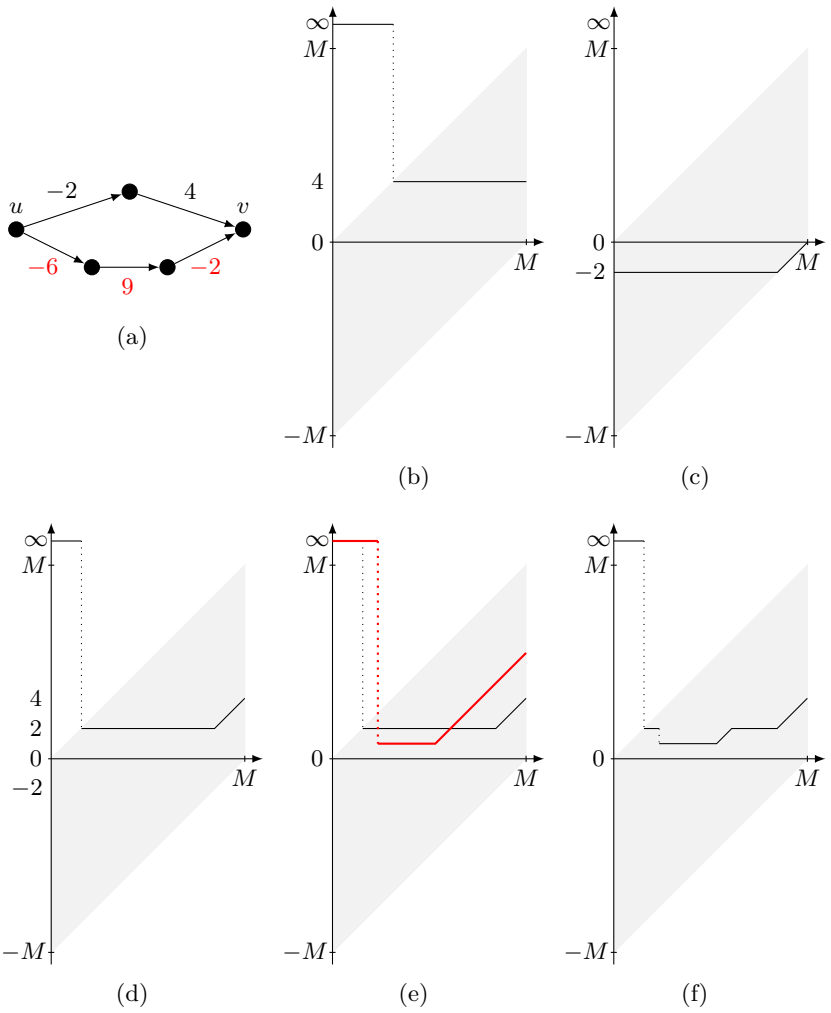

Fig. 1. An example of link and merge operations for energy consumption cost functions: (a) the original graph with constant energy consumption values; (b) the function induced by the arc with weight 4, mapping SoC onto consumption; the shaded area depicts those consumption values that fulfill the battery constraints; (c) the function induced by the arc with weight -2; (d) the resulting cost function after linking the black path; (e) the cost functions of both the red and the black path; (f) the result after merging—note that this function also represents the u–v profile.

optimal s–t paths for *every* value of $b_s \leq M$. (Note that this corresponds to the s–t consumption function.) It is a required preprocessing ingredient to many speedup techniques (such as ours) and helpful for deciding how much to charge the battery before departing.

3 Basic Approach

Our baseline algorithm to compute SoC queries is a (label-correcting) variant of Dijkstra's algorithm, called LC. For every vertex v, it maintains a label $\mathrm{lbl}(v)$, initially set to ∞, except $\mathrm{lbl}(s)$, which is set to 0 (consumption). A priority queue Q is initialized with source vertex s and key $\mathrm{lbl}(s)$. The main loop *scans* the vertex u of minimum key $\mathrm{minKey}(Q)$ by extracting it from the queue. Then, for each arc $a = (u, v)$, the algorithm evaluates its cost function f_a for SoC $b(u) = b_s - \mathrm{lbl}(u)$. If $\mathrm{lbl}(u) + f_a(b(u)) < \mathrm{lbl}(v)$, it *relaxes* the arc by decreasing $\mathrm{lbl}(v)$ and updating the queue, accordingly. The algorithm stops as soon as the queue runs empty. Then,

for each vertex v, $\mathrm{lbl}(v)$ provably holds the minimum energy consumption necessary to reach v when starting at s with SoC b_s. Note that correctness follows from [17].

If arc costs are negative, the algorithm is label-*correcting*: It may scan vertices more than once, if their labels are improved via sub-paths of negative length. It is well-known that this might trigger (exponentially many) rescans over large parts of the graph. However, this is only the case if negative shortest paths have a large positive prefix (in relation to the graph diameter), which is unlikely in our scenario. Also note that our inputs do not contain negative cycles due to physical constraints.

To get rid of negative costs completely, we also consider three potential shifting methods, which re-enable (label-setting) Dijkstra's algorithm. The first variant, Dij-PV, uses a potential induced by an arbitrary fixed vertex v^*. It takes v^* to set $\Pi(v) = \underline{f}_P$, where P is a minimum cost v^*–v path. To compute these values, we run LC (once) evaluating the lower bound $\underline{f}_a$ of arc costs. The second variant, Dij-PS, runs multi-source LC to obtain potentials. Unlike Dij-PV, it initially sets vertex labels to 0, and adds all vertices to the priority queue that have incident arcs with negative cost (similarly to [25]). Finally, Dij-PH uses a height-induced potential $\Pi(v) = \gamma \cdot h(v)$. Unlike [28], who use a specific physical energy consumption model to define γ, we determine γ by a single linear scan over all arcs $a = (u, v)$, maintaining a lower bound $\underline{\gamma}$ and an upper bound $\overline{\gamma}$. Since $\underline{f}_a + \gamma h(u) - \gamma h(v) \geq 0$ must hold, downhill arcs update $\underline{\gamma}$, which is set to $\underline{\gamma} = \max(\underline{\gamma}, \underline{f}_a/(h(v) - h(u)))$. Uphill arcs update $\overline{\gamma} = \min(\overline{\gamma}, \underline{f}_a/(h(v) - h(u)))$. If, in the end, $\underline{\gamma} \leq \overline{\gamma}$ holds, $\underline{\gamma}$ yields a feasible potential (which is always the case for our experiments).

We apply the following *stopping criteria*: Dij-x stops, if the algorithm scans the target vertex t. For LC, since it is label-correcting, this cannot be applied. However, by precomputing the minimum shortest (possibly negative) path length π, it now checks $\mathrm{minKey}(Q) + \pi > \mathrm{lbl}(t)$. One could compute π by connecting a super source s' and sink t' with all vertices of G that have incident arcs with negative cost, and then run an LC s'–t' query. However, in practice, the following method turned out to be faster (by factor 2–3). For every vertex, it runs an LC query that uses $\underline{f}_a$ as arc cost. It minimizes the minimum shortest path length whenever it scans a vertex. Arcs $a = (u, v)$ are only relaxed if they improve the head label $\mathrm{lbl}(v)$ to a negative value. We stop each query if $\mathrm{minKey}(Q) \geq 0$ (the minimum negative path cannot have a positive prefix). We do not reinitialize vertex labels between queries, since a search can be pruned if a path with shorter prefix was found by a previous query.

3.1 Profile Queries

Although computationally more expensive, profile search is conceptually easy using cost functions: Following [6], the algorithm maintains, for each vertex v, a label f_v that represents an s–v consumption profile (taking the form of a cost function; recall Fig. 1). It initializes $f_v = f^\infty$ for all $v \in V$, except $f_s = f^0$, and adds s to the priority queue Q with $\mathrm{key}(s) = 0$. In the main loop, the algorithm scans the vertex u of minimum key by extracting it from Q and scans its incidents arcs $a = (u, v)$; It computes $f_v^{\mathrm{new}} = \mathrm{link}(f_u, f_a)$ and sets $f_v = \mathrm{merge}(f_v, f_v^{\mathrm{new}})$, updating $\mathrm{key}(v)$ if necessary. As key we use the lower bound of a vertex label, i. e., $\mathrm{key}(v) = \underline{f}_v$. Note that this approach is label-correcting (even after applying potential shifting) and that its performance depends on the complexity of the cost functions [6]. We apply the following *target pruning* rule: Given any cost function f, let $\underline{b}_f \in [0, M]$ denote the smallest SoC value, for which $f(\underline{b}_f)$ is finite. Then, whenever the algorithm scans a vertex v, it checks if both $\underline{b}_{f_v} \geq \underline{b}_{f_t}$ and $\underline{f}_v \geq \overline{f}_t$ hold. If that is the case, the algorithm *prunes* the search at vertex v, i. e., it does not scan any outgoing arcs of v.

3.2 Implementation Details

In order to achieve best performance in practice, an efficient implementation of piecewise linear functions is crucial. Because of their specific form (discontinuous; slope in $\{0, 1\}$), we implement cost functions by storing pairs of point and slope. Since, in practice, our functions f consist of few interpolation points on average, it is best to evaluate $f(b)$ by a linear scan. As an optimization we use a *compressed function representation*. It stores a single 32-bit integer for functions that have simple form (explicitly checking for battery constraints in the algorithm). Following [12], we implement link and merge by linear scans in general. However, for compressed functions these operations are much simpler: They basically reduce to scalar additions/minimums, and some border cases. To improve spatial locality of the profile search, we store vertex labels $f_{v_1}, \ldots, f_{v_n}$ as a dynamic adjacency array. For each label, it uses a flag to indicate if f_v is a compressed function, storing the (compressed) value directly at the label. If not, it stores (bit-compressed) indices to an *interpolation point array*. Note that the number of interpolation points of f_v may vary during the algorithm. We mark empty slots in the array in order to make efficient (re-)use of space. Our implementation for the experiments includes all of the improvements.

4 Overlays

This section introduces our acceleration technique, which extends the *Customizable Route Planning* (CRP) approach, introduced by Delling et al. [8], exploiting ideas from [10, 24, 26, 30, 31]. It uses three phases: A (potentially costly) offline metric-independent *preprocessing phase*, a metric-dependent preprocessing phase, called *customization phase*, and, finally, the (online) *query phase*. Its main strength is that customization is very quick: A new metric can be incorporated in a few seconds, even on continental networks, while a single arc cost can be updated in only a few microseconds [8]. This is particularly important in our context, since energy consumption of electric vehicles may vary with, for example, vehicle load and changing weather conditions, or due to newly learned consumption data [14]. In the following, we recap the CRP algorithm, describing our extensions along the way.

Preprocessing The preprocessing phase of CRP computes a multilevel *overlay* [26, 31] of the input graph $G = (V, A)$. An overlay is a graph $G' = (V' \subseteq V, A')$, such that distances between vertices of G' are the same as in G. They are obtained from a nested multilevel partition $(\mathcal{C}^1, \ldots, \mathcal{C}^L)$, as follows. For a fixed level ℓ, the overlay graph of level ℓ consists of exactly the boundary vertices of $\mathcal{C}^\ell$. Besides boundary arcs of G (with respect to $\mathcal{C}^\ell$), it also contains for each cell $C \in \mathcal{C}^\ell$ and all pairs of boundary vertices $u, v \in C$ an arc (u, v). This results in a full clique of arcs over the cell's boundary vertices. Similarly to [8], 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 boundary vertices of $\mathcal{C}^1$). Only clique arcs are kept in a separate data structure per level, and they are organized as matrices of preallocated contiguous memory. (Note that boundary arcs are already present in the input graph.) In contrast to [8], we *reorder* the vertices of G, such that overlay vertices are pushed to the front (order by descending level), breaking ties by cell. Non-overlay vertices are ordered by their level-1 cells. This improves spatial locality for customization and query, and simplifies mapping between overlay and original vertices. Preprocessing must only be rerun if the *topology* of the input changes (significantly). Since this happens infrequently in practice, somewhat higher preprocessing times are not an issue.

Customization The customization phase uses the output of the preprocessing phase to compute the metric of the overlays, i. e., for each clique arc it must compute its cost function. It proceeds in a bottom-up fashion, starting with the lowest level. Within level ℓ, each cell $C \in \mathcal{C}^\ell$ is

processed independently. A cell C is processed by running, for each boundary vertex $u \in C$, a *profile search* (cf. Section 3.1) from u. The search is, thereby, restricted to cell C, i. e., it does not relax any arcs pointing outside C. At every boundary vertex $v \in C$, this results in a cost function f_v, which is assigned to the clique arc (u, v) of cell C. Customization can be parallelized by distributing different cells (on a level) among processors. In contrast to [8], the complexity of the cost functions is not known in advance. In fact, our overlay uses a (single) dynamic adjacency array to store interpolation points of clique arcs. 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 clique arc functions it has computed. These logs are sequentially merged at the end of processing level ℓ. However, preliminary experiments indicate that more than 80 % of the cost functions can be compressed. Only for the remaining cases a thread uses its log; compressed functions are written to the overlay directly. Unlike the preprocessing phase, customization is much faster, taking mere seconds in practice (cf. Section 5). Note that, like [8], when processing level $\ell + 1$, we make use of the (already computed) overlay of level ℓ, which significantly improves customization time.

SoC Query For vertices s, t, and initial SoC b_s, the query operates on a search graph consisting of, (a), the overlay graph of the topmost level L, (b), all cells from the overlay that contain s or t, and (c), the subgraph of the original graph induced by the level-one cells that contain s or t. Then, any algorithm from Section 3 can be run on this search graph to get provably optimal solutions. Also, potentials computed for the original graph naturally carry over to the overlays. From now on, we assume that potentials are always available, and refer to the algorithm as *Unidirectional Multi-Level-Dijkstra* (Uni-MLD). Note that we do not need to explicitly extract the search graph. Instead, the level and cell on which Uni-MLD scans arcs are determined implicitly from the partition data [8]. To obtain the full path description, clique arcs a on level ℓ can be unpacked, by (recursively) running a local query on the overlay of level $\ell - 1$, restricted to the cell of a [8].

Bidirectional Search A common technique to accelerate queries is *bidirectional search*. Basically, it simultaneously runs a *forward search* from s and a *backward search* from t until a stopping condition is met. Thereby, the algorithm minimizes a *tentative distance* value μ (initialized to infinity), whenever the searches meet at a vertex m. After stopping, the shortest path of length μ is obtained by concatenating $P_{s,m}$ with $P_{m,t}$. Unfortunately, the final SoC at t is not known in advance, which prevents running a regular SoC backward search. Therefore, we present two approaches that augment [12] the backward search: *Bidirectional Profile-Evaluating Multi-Level-Dijkstra* (BPE-MLD) and *Bidirectional Distance-Bounding Multi-Level-Dijkstra* (BDB-MLD). Both use a regular SoC forward search.

The first (straightforward) approach, BPE-MLD, runs a *backward profile search* (cf. Section 3.1) from t, which does not require an initial SoC value, computing v–t profiles f_v as vertex labels. Whenever either search scans an arc toward a vertex v that has already been touched by the opposite search, it evaluates the profile f_v (obtained from the backward search) at SoC $b(v)$ (obtained from the forward search as $b(v) := b_s - \mathrm{lbl}(v)$), thereby, minimizing $\mu := f_v(b(v))$. The algorithm may stop as soon as *any* path it may still find has cost either exceeding μ or the battery capacity M. Recall from Section 3.1 that the (backward) profile search uses lower bounds (of its vertex labels) as keys in its priority queue Q_B. Hence, we stop the search as soon as $\mathrm{minKey}(Q_F) + \mathrm{minKey}(Q_B) > \min(\mu, M)$ 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 this is similar to [20]. However, we have to carefully account for battery constraints. To do so, the backward search maintains, for every

vertex v, three labels: Lower and upper bounds on the cost from v to t, denoted $\underline{c}(v)$ and $\overline{c}(v)$, and a minimum battery SoC to reach t, denoted $\underline{b}(v)$. We define $\overline{c}(v)$ consistently with $\underline{b}(v)$: An SoC of $\underline{b}(v)$ implies that t can be reached from v with cost at most $\overline{c}(v)$. Labels are initially set to infinity, except at t, for which they are set to $\underline{c}(t) = \overline{c}(t) = \underline{b}(t) = 0$. The backward search is then running Dijkstra on $\underline{c}$. When scanning an arc $a = (u, v)$, it uses $\underline{f}_a$ as metric. Simultaneously (during the same arc scan), $\overline{c}$ and $\underline{b}$ are computed as follows. Let $b \in [0, M]$ be the minimum SoC that is necessary to traverse a, i. e., the smallest value for which f_a is finite. Then the minimum SoC $\underline{b}(u)$ to travel from u to t is determined by the minimum of b itself and the cost $f_a(b)$ of traversing a with SoC b plus $\underline{b}(v)$ (the minimum SoC to get from v to t). On the other hand, the maximum cost at u is determined by $\overline{c}(v) + \overline{f}_a$. Summarizing, whenever the algorithm scans arc $a = (u, v)$, it checks $\max(\underline{b}(v) + f_a(b), b) \leq \underline{b}(u)$ and $\overline{c}(v) + \overline{f}_a \leq \overline{c}(u)$, updating $\underline{b}(u)$ and $\overline{c}(u)$, if necessary. The forward search now minimizes an upper bound μ on the cost of the shortest s–t path. Whenever it scans a vertex v with SoC level $b(v)$, it checks if $b(v) \geq \underline{b}(v)$. Only in this case, it tries to update μ by setting $\mu = \min(\mu, b(v) + \overline{c}(v))$. Moreover, it (independently from the previous check) *prunes* the search at v (i. e., it does not scan outgoing arcs from v), if either v was settled by the backward search and $b(v) + \underline{c}(v) \geq \min(\mu, M)$, or $\text{minKey}(Q_F) + \text{minKey}(Q_B) \geq \min(\mu, M)$ holds. The algorithm stops when the forward search reaches t and determines the cost $b(t)$.

Parallelization To get additional speedup, we propose parallelizing the query in a multicore scenario. We assign different processors to forward and backward search, where they run independently. To minimize μ, 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 be safely ignored: The correct value μ 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 μ 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, 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 improves μ. Note that this list is small in practice.

Reachability Flags To accelerate long-range queries for which the target is unreachable, we may additionally precompute *reachability flags*: For the topmost level L of the partition, we keep a bit matrix, whose entry i, j is set iff cell C_j is *reachable* from cell C_i. More precisely, we run, for each cell C_i (in parallel), a Dijkstra on the level-L overlay from all boundary vertices of cell C_i. It uses lower bounds $\underline{f}_a$ as costs. It sets all flags i, j of the matrix for which there exists a boundary vertex of cell C_j at distance at most M. 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. In practice, storing these bits requires little additional space.

5 Experiments

We implemented all algorithms in C++ using g++ 4.5.1 (flag -O3) as compiler. Experiments were conducted on a dual 8-core Intel Xeon E5-2670 clocked at 2.6 GHz, with 64 GiB of DDR3-1600 RAM, 20 MiB of L3 and 256 KiB of L2 cache. Unless otherwise noted, we ran our implementation sequentially.

Input Data and Methodology Our main test instances are based on the road network of Europe, kindly provided by PTV AG. Road segments have associated average speeds and road categories. We obtained height information for the vertices from the freely available NASA Shuttle Radar

Topography Mission[1] (SRTM) data. It covers large parts of the world with a horizontal resolution of approximately 90 meters. We filled (rarely) missing data points by interpolating from neighbors.

Our energy consumption data stems from PHEM (Passenger Car and Heavy Duty Emission Model), developed by the Graz University of Technology [22]. PHEM is a micro scale emission model based on backwards longitudinal dynamics simulation. Beside other applications, PHEM is used to calculate emissions for passenger cars, heavy and light duty vehicles for the Handbook Emission Factors (HBEFA) [23]. The HBEFA driving cycles cover a large variety of road categories, speed limits, traffic situations and slopes. These cycles were calculated using different EV configurations and vehicle types to generate average energy consumption values for all available driving situations. We carefully mapped them to our network by a heuristic that measures the similarity between road segments of the PTV data and the parameters of PHEM. Finally, we removed all vertices from the graph where either no height data is available (not even via sensible interpolation), or which cannot be mapped to a PHEM road category (such as "private" roads). We extracted the largest strongly connected component of the European network (22 198 628 vertices, 51 088 095 arcs).

For our experiments, we use two instances of the PHEM data. The first, Europe PG-16, is based on a Peugeot iOn with a battery capacity of 16 kWh, applied to the European network. The second, Europe EV-85, is an artificial electric vehicle model [36], for which we assume a larger battery capacity of 85 kWh (similar to that of the current high-end Tesla model). To get the best possible cruising range, we disabled auxiliary consumers in both instances. 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 arcs with negative cost is 11.8 % and 15.2 % for Europe PG-16 and Europe EV-85, respectively.

As partitioning tool we used PUNCH [9], which is explicitly developed for road networks and aims at minimizing the number of boundary arcs. For Europe, we use a 4-level partition with maximum cell sizes of 2^6, 2^{10}, 2^{14}, 2^{18} vertices (values determined by preliminary experiments). Computing the partition took 24 minutes. Considering that the road topology changes rarely (i. e., the partition needs be updated only when roads are built or (permanently) closed), this is sufficiently fast in practice. For a detailed evaluation of the tradeoff between partitioning speed and quality, see [9].

Evaluating Algorithms Table 1 reports figures for our algorithms on both Europe PG-16 and Europe EV-85. Note that since the range of the electric vehicles 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. Instead, we generate queries by first picking a random source vertex s from which we run a preliminary query with initial SoC set to 110% of the vehicle's battery capacity (to still generate some out-of-range queries). The target vertex t is then picked randomly from the search space of that query. Accordingly, we do not use reachability flags for the experiments of Table 1.

We ran 10 000 queries for the MLD algorithms, and 1 000 (a subset) for LC, Dijkstra and the profile query. SoC queries assume full battery capacity at s. Less capacity would only result in faster query times. We report the number of vertex (re)scans and the query time (in milliseconds). Customization time (in seconds) depicts, for the label-correcting algorithm LC, the time to compute the minimum path distance used to enable the stopping criterion (sc.); for all Dijkstra-based variants, the time to compute potentials; and for the MLD variants, the time for both metric customization and computing potentials. Finally, we report the total amount of (metric-dependent) space overhead in bytes per vertex (B/n).

[1] `http://www2.jpl.nasa.gov/srtm/`

Table 1. Evaluating our algorithms on both vehicle instances: A Peugeot iOn with a 16 kWh battery (Europe PG-16) and an artificial vehicle with a 85 kWh battery (Europe EV-85). The column "sc." indicates whether a stopping criterion or target pruning rule is applied (•) or not (◦). We also report figures on an instance from [34]: It uses the geographical distance and height difference of the arcs to model consumption and assumes unlimited capacity.

		Europe PG-16				**Europe EV-85**				**Japan DH-∞**			
		CUSTOM.		QUERIES		CUSTOM.		QUERIES		CUSTOM.		QUERIES	
Algorithm	sc.	space [B/n]	time [s]	vertex scans	time [ms]	space [B/n]	time [s]	vertex scans	time [ms]	space [B/n]	time [s]	vertex scans	time [ms]
LC	◦	—	—	393 833	54.5	—	—	4 471 230	709.6	—	—	26 078 917	3 761.8
LC	•	0.0	3.86	333 272	46.9	0.0	5.20	3 001 014	486.6	0.0	3.70	13 133 641	1 939.0
Dij-PH	•	4.0	0.69	213 765	29.4	4.0	0.70	2 359 140	380.6	4.0	—	12 933 517	1 933.0
Dij-PV	•	4.0	3.84	221 368	25.5	4.0	3.91	2 361 997	288.0	4.0	3.89	13 048 081	994.4
Dij-PS	•	4.0	8.37	214 666	30.4	4.0	11.01	2 360 646	379.6	4.0	3.01	12 933 031	1 984.8
Prof-PH	•	4.0	0.69	260 540	56.6	4.0	0.70	2 904 764	741.2	4.0	—	16 208 904	3 203
Uni-MLD-PH	•	13.6	4.32	941	0.5	14.5	5.12	2 410	1.9	7.7	2.06	2 205	1.0
BPE-MLD-PH	•	13.6	4.32	929	0.3	14.5	5.12	2 266	1.4	7.7	2.06	2 198	0.8
BDB-MLD-PH	•	13.6	4.32	1 203	0.3	14.5	5.12	2 917	1.1	7.7	2.06	2 711	0.7
Dij-PH [34]	•	—	—	—	—	—	—	—	—	4.0	—	14 431 809	6492.6
EVCH [34]	•	—	—	—	—	—	—	—	—	23.0	14 329.87	10 024	44.9
EVCH (scaled)	•	—	—	—	—	—	—	—	—	23.0	8 791.33	10 024	13.4

It is not surprising that the label-correcting approach is the slowest on both instances. Although the number of vertex rescans is comparatively low (less than 10 %; not shown in the table), the rather weak stopping criterion, which has to add an offset to the tentative distance, induces a larger search space size. However, by comparing the LC variants, we observe that computing this offset already amortizes after 23 queries on average for EV-85.

The different variants of vertex potentials yield similar query times. Somewhat surprisingly, Dij-PV has the best query times, however, with higher variance (available from Fig. 2; the lower average value is mostly induced by long-range queries, see below for a detailed discussion). Since Dij-PH has by far the smallest customization time, we use height-induced potentials for the profile query and all MLD variants. Profile queries admit—in contrast to time-dependent route planning [12]—practical running times in our scenario: We observe a slow down by a factor of less than two over LC.

Regarding MLD, all variants provide query times of below two milliseconds on both European instances. Customization only takes about five seconds (when parallelized), enabling very frequent metric updates for the whole network in a server scenario; 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). Bidirectional search yields another, yet small, speedup. We observe that BDB-MLD slightly outperforms BPE-MLD (by about 15 % on average). Note that depending on the application, bidirectional search might not pay off: It is run on two cores but the speedup achieved is less than two. Also note that customization times of Europe EV-85 are higher than on Europe PG-16. We attribute this to the larger number of negative arcs in the former instance. Space consumption is dominated by storing interpolation points. One can reduce it by removing the lowest level of the partition for queries (keeping it only to accelerate customization) [8]: On both European instances this saves space by a factor of two, while queries are slowed down by only about 10 % on average (not reported in the table). Furthermore, note that for all x-PH variants, we include 4 B/n space overhead for storing the height-induced potential $\Pi(v) = \gamma \cdot h(v)$ at each vertex. If height values are assumed to be already available as part of the input, we

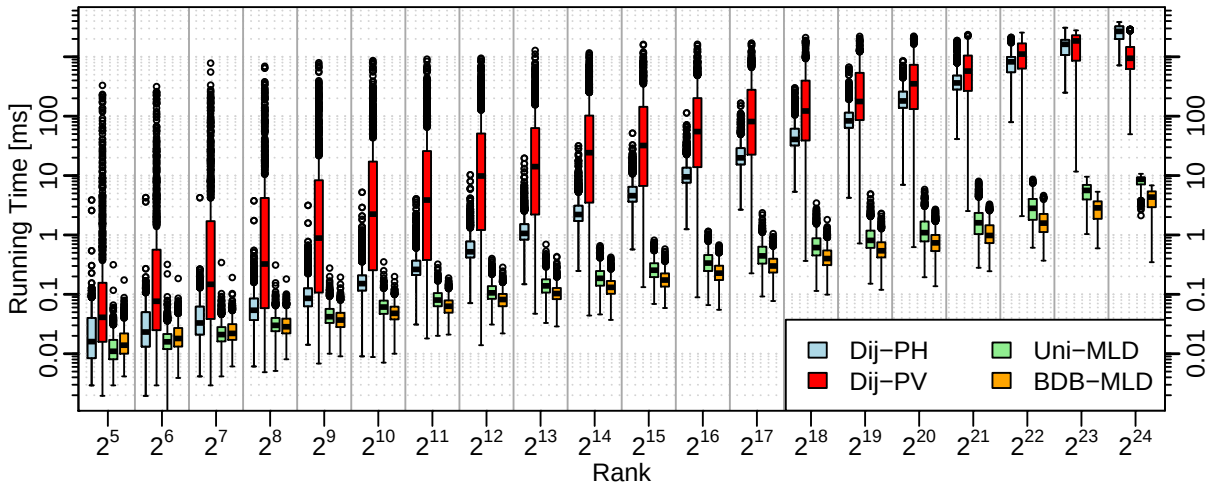

Fig. 2. Running times subject to Dijkstra rank for our algorithms on the Europe EV-1000 instance. Smaller ranks indicate more local queries. Reachability flags for the MLD algorithms are disabled and battery capacity is increased to the point where it does not constrain vehicle range.

could just store the single value γ, and compute $\Pi(v)$ on-demand in the algorithm. This would reduce customization space to about 10 B/n on both European instances.

Evaluating Scalability Fig. 2 analyzes the scalability of our algorithms on the Europe EV-x instance using the Dijkstra rank method [11]. Because of negative arc costs, we use the label-correcting algorithm for computing ranks: It orders the vertices by the time they were last extracted from the priority queue, from which ranks are determined. Moreover, to get meaningful results, we *increase battery capacity* from 85 to 1 000 kWh (which corresponds to a cruising range of roughly 5 000 km, enough to make all our random queries in-range). We report Dij-PH, Dij-PV, Uni-MLD-PH, and BDB-MLD-PH. Values for each rank are obtained from 1 000 random queries. We observe that our MLD-based algorithms are consistently faster than Dijkstra's algorithm. Except for very local queries (below rank 2^8), bidirectional search always pays off. Interestingly, Dij-PV has much higher variance compared to Dij-PH, and significantly lower query times for long-range queries. Recall that potentials of Dij-PV are determined by distances from a single vertex, which seems to result in a highly distorted search space. We conclude that using our most sophisticated method, BDB-MLD-PH, we achieve average (respective maximal) query times of under 4.2 ms (6.9 ms), for any rank value, while Uni-MLD-PH still stays below 7.9 ms (10.7 ms).

Fig. 3 shows running times subject to Dijkstra rank for the instance Europe PG-16. In contrast to Table 1 and Fig. 2, we enable reachability flags, keep battery capacity at 16 kWh, and do not constrain query distance. Hence, the plot shows the effect of the flags on long-distance queries for MLD variants. Starting with rank 2^{19}, query times drop gradually. Above rank 2^{22}, the target is almost never reachable, yielding query times of under 0.01 ms on average. 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 took less than 100 ms. Similarly to Fig. 2, Dij-PV is consistently slower and has a higher variance than Dij-PH in most cases, except for high ranks.

Comparison to Contraction Hierarchies At the time of writing, the fastest available approach that computes energy-optimal routes for electric vehicles is based on Contraction Hierarchies [18, 34]. To compare our algorithms with that method, we also ran experiments on the largest instance used in [34], which was kindly given to us by the authors. In the following, we refer to their approach as EVCH (as in Electric Vehicle Contraction Hierarchies) and to their instance

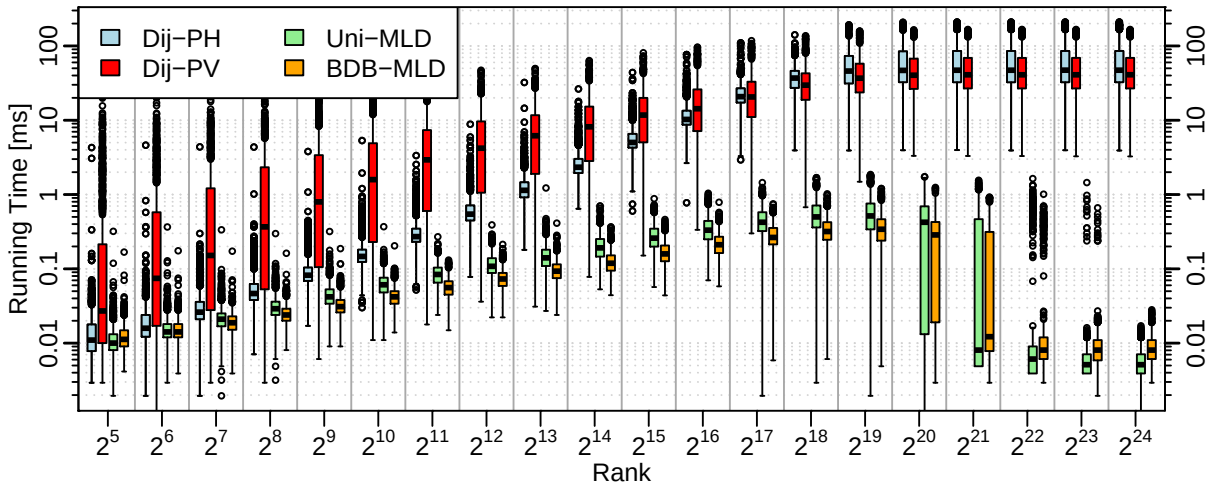

Fig. 3. Running times subject to Dijkstra rank for our algorithms on the Europe PG-16 instance. We enable reachability flags for the MLD algorithms. As in Fig. 2, smaller ranks indicate more local queries.

by Japan DH-∞. It is based on an OpenStreetMap (OSM) export of the Japan road network, augmented with SRTM data. It has 25 970 678 vertices and 54 141 580 arcs. Note that these figures are slightly higher than for our European instance, however, OSM networks are notorious for having exceptionally many vertices of low degree that (only) model geometry. To account for that, we use a 4-level partition with increased maximum cell sizes of 2^7, 2^{11}, 2^{15}, 2^{19} vertices. Using PUNCH [9], computing the partition took less than half an hour.

Instead of detailed realistic vehicle data the instance uses a simple consumption model: Given an arc $a = (u, v)$, its cost is assumed to depend on horizontal distance and vertical heights, only. More precisely, $c_a = \kappa \cdot \text{dist}(a) + \lambda \cdot (h(v) - h(u))$, if $h(v) - h(u) \geq 0$; $c_a = \kappa \cdot \text{dist}(a) + \alpha \cdot (h(v) - h(u))$, otherwise. According to [34], we set $\kappa = 0.02$, $\lambda = 1$, $\alpha = 0.25$, and assume very large battery capacity. As a result, all queries are in-range, and reachability flags are disabled in our algorithm. The amount of arcs with negative weight is 9.0 %, which is less than for Europe (even though Japan is comparably mountainous). The reason for this is the simpler consumption model, given above parameters. Note that, when applying this model to the European network, we observe that the amount of negative arcs also drops from 15.2 % for EV-85 to only 4.4 %.

Table 1 also reports results on Japan DH-∞. It shows figures for the Dijkstra and EVCH implementations reported in [34], where a height-induced potential follows from the model (i. e., $\gamma = \alpha$). Therefore, it does not require any customization time. Since these results were obtained on slower machines, we report scaled timings (the bottom row of the table). We know of two typical approaches to scale running times between machines: (a) Using running times of a common baseline algorithm, and (b), having access to the same hardware for scaling experiments. Unfortunately, the authors of [18, 34] use more than one machine (an AMD Opteron 6172 with 2.1 GHz for preprocessing, and an Intel i3-2310M with 2.1 GHz for queries) without reporting scaling factors themselves. Therefore, we have to resort to both scaling approaches. For EVCH's query time, we scale very conservatively based on their and our Dij-PH implementation (assuming that both implementations are equally well engineered). This maintains their reported speedup. Since we have an Opteron 6172 available, scaling EVCH's preprocessing time is done by our own scaling experiment. Although not specifically mentioned in [18, 34], we infer that their Dijkstra implementation uses a stopping criterion: The search space reported in [34] is about 56 % of the graph size.

At first glance, Japan DH-∞ seems to be harder than Europe: All LC and Dij-x variants scan more vertices and have higher query times (Dij-PV still being consistently faster), which is

due to the larger graph size and larger range of (random) queries. However, we observe that the MLD variants all consistently perform better on Japan DH-∞ than on Europe. Recall that OSM modeling overhead has an impact only for the lowest level of the partition.

In comparison to EVCH, our query is faster by an order of magnitude, while having a factor of 4–5 smaller search space size. At the same time, our (metric-dependent) preprocessing (customization) is faster by more than a factor of 4 000, while requiring a third of the space. Even when run on a single core, which still only requires 20.98 s, customization of MLD is more than 400 times as fast as EVCH preprocessing. Note that customizing a single arc only requires 18.7 ms for our approach (not reported in the table).

Clearly, some of these performance differences might be due to differently tuned implementations. Yet, our findings add to previous observations that separator-based approaches are more robust for non-traveltime metrics than Contraction Hierarchies [8].

6 Final Remarks

In this paper we studied computing energy-optimal routes for electric vehicles, for which key challenges include negative costs (recuperation), battery capacity constraints, and frequently changing metrics. Based on the Customizable Route Planning (CRP) approach, our presented algorithms handle these challenges in a sound manner. In particular, we introduced an algorithm to compute profile queries (a necessary ingredient to CRP), and a nontrivial adaption of bidirectional search. Experiments using real world consumption data on the continental network of Europe indicate that our approach incorporates new metrics within seconds, after which queries can be answered in 1.1 ms or less, on average—making it the fastest available technique for electric vehicle route planning.

Future work includes integrating turn costs (in terms of energy consumption) and recent ideas from [13] into our algorithm. Also, building on results from [33], we are interested in adding more criteria, such as travel time, to our approach (e. g., via Pareto-optimization) to trade off speed and energy consumption (electric vehicles consume less energy when driven slower). We are also interested in adapting to mobile dynamic scenarios [19]. Furthermore, re-applying ideas from this work to EVCH might be worthwhile.

Finally, we thank Raphael Luz for providing the consumption data [23, 36], Renato Werneck for running PUNCH [9], and Moritz Kobitzsch for interesting discussions. We also thank Christian Schulz and Dennis Luxen for providing BUFFOON [29] and OSRM [27], respectively, which we used in our preliminary experiments. We thank Sabine Storandt for making Japan DH-∞ available to us, and Konstantinos Demestichas for providing sample data on energy consumption of electric vehicles.

References

1. A. Artmeier, J. Haselmayr, M. Leucker, and M. Sachenbacher. The Shortest Path Problem Revisited: Optimal Routing for Electric Vehicles. In R. Dillmann, J. Beyerer, U. D. Hanebeck, and T. Schultz, editors, *Proceedings of the 33rd Annual German Conference on Advances in Artificial Intelligence*, Lecture Notes in Computer Science 6359, pp. 309–316. Springer, September 2010.
2. D. A. Bader, H. Meyerhenke, P. Sanders, and D. Wagner, editors. *Graph Partitioning and Graph Clustering: Tenth DIMACS Implementation Challenge*, DIMACS Book 588. American Mathematical Society, 2013.
3. G. V. Batz, R. Geisberger, P. Sanders, and C. Vetter. Minimum Time-Dependent Travel Times with Contraction Hierarchies. *ACM Journal of Experimental Algorithmics*, 18(1.4):1–43, April 2013.
4. R. Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.
5. W. Burgard and D. Roth, editors. *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*. AAAI Press, August 2011.
6. B. C. Dean. Continuous-Time Dynamic Shortest Path Algorithms. Master's thesis, Massachusetts Institute of Technology, 1999.

7. B. C. Dean. Shortest Paths in FIFO Time-Dependent Networks: Theory and Algorithms. Technical report, Massachusetts Institute Of Technology, 2004.
8. D. Delling, A. V. Goldberg, T. Pajor, and R. F. Werneck. Customizable Route Planning. In P. M. Pardalos and S. Rebennack, editors, *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, Lecture Notes in Computer Science 6630, pp. 376–387. Springer, 2011.
9. D. Delling, A. V. Goldberg, I. Razenshteyn, and R. F. Werneck. Graph Partitioning with Natural Cuts. In *25th International Parallel and Distributed Processing Symposium (IPDPS'11)*, pp. 1135–1146. IEEE Computer Society, 2011.
10. D. Delling, M. Holzer, K. Müller, F. Schulz, and D. Wagner. High-Performance Multi-Level Routing. In C. Demetrescu, A. V. Goldberg, and D. S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, DIMACS Book 74, pp. 73–92. American Mathematical Society, 2009.
11. D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering Route Planning Algorithms. In J. Lerner, D. Wagner, and K. A. Zweig, editors, *Algorithmics of Large and Complex Networks*, Lecture Notes in Computer Science 5515, pp. 117–139. Springer, 2009.
12. D. Delling and D. Wagner. Time-Dependent Route Planning. In R. K. Ahuja, R. H. Möhring, and C. Zaroliagis, editors, *Robust and Online Large-Scale Optimization*, Lecture Notes in Computer Science 5868, pp. 207–230. Springer, 2009.
13. D. Delling and R. F. Werneck. Faster Customization of Road Networks. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, Lecture Notes in Computer Science 7933, pp. 30–42. Springer, 2013.
14. K. Demestichas, M. Masikos, E. Adamopoulou, S. Dreher, and A. D. de Arkaya. Machine-Learning Methodology for Energy Efficient Routing. In *the 19th World Congress on Intelligent Transport Systems*, October 2012.
15. N. Deo and C. Pang. Shortest-path algorithms: Taxonomy and annotation. *Networks*, 14(2):275–323, 1984.
16. E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.
17. S. E. Dreyfus. An Appraisal of Some Shortest-Path Algorithms. *Operations Research*, 17(3):395–412, 1969.
18. J. Eisner, S. Funke, and S. Storandt. Optimal Route Planning for Electric Vehicles in Large Network. In Burgard and Roth [5].
19. R. Geisberger, P. Sanders, D. Schultes, and C. Vetter. Exact Routing in Large Road Networks Using Contraction Hierarchies. *Transportation Science*, 46(3):388–404, August 2012.
20. R. J. Gutman. Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX'04)*, pp. 100–111. SIAM, 2004.
21. P. E. Hart, N. Nilsson, and B. Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.
22. S. Hausberger. Simulation of Real World Vehicle Exhaust Emissions, 2003.
23. S. Hausberger, M. Rexeis, M. Zallinger, and R. Luz. Emission Factors from the Model PHEM for the HBEFA Version 3. Technical Report I-20/2009, University of Technology, Graz, 2009.
24. M. Holzer, F. Schulz, and D. Wagner. Engineering Multilevel Overlay Graphs for Shortest-Path Queries. *ACM Journal of Experimental Algorithmics*, 13(2.5):1–26, December 2008.
25. D. B. Johnson. Efficient Algorithms for Shortest Paths in Sparse Networks. *Journal of the ACM*, 24(1):1–13, January 1977.
26. S. Jung and S. Pramanik. An Efficient Path Computation Model for Hierarchically Structured Topographical Road Maps. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1029–1046, September 2002.
27. D. Luxen and C. Vetter. Real-Time Routing with OpenStreetMap data. In *Proceedings of the 19th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM Press, 2011.
28. M. Sachenbacher, M. Leucker, A. Artmeier, and J. Haselmayr. Efficient Energy-Optimal Routing for Electric Vehicles. In Burgard and Roth [5].
29. P. Sanders and C. Schulz. Distributed Evolutionary Graph Partitioning. In *Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX'12)*, pp. 16–29. SIAM, 2012.
30. F. Schulz, D. Wagner, and K. Weihe. Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. *ACM Journal of Experimental Algorithmics*, 5(12):1–23, 2000.
31. F. Schulz, D. Wagner, and C. Zaroliagis. Using Multi-Level Graphs for Timetable Information in Railway Systems. In *Proceedings of the 4th Workshop on Algorithm Engineering and Experiments (ALENEX'02)*, Lecture Notes in Computer Science 2409, pp. 43–59. Springer, 2002.
32. C. Sommer. Shortest-Path Queries in Static Networks, 2012. Submitted. Preprint available at `http://www.sommer.jp/spq-survey.htm`.
33. S. Storandt. Quick and energy-efficient routes: computing constrained shortest paths for electric vehicles. In *Proceedings of the 5th ACM SIGSPATIAL International Workshop on Computational Transportation Science*, pp. 20–25. ACM Press, 2012.
34. S. Storandt. *Algorithms for vehicle navigation.* PhD thesis, Universität Stuttgart, February 2013.

35. W. J. Sweeting, A. R. Hutchinson, and S. D. Savage. Factors affecting electric vehicle energy consumption. *International Journal of Sustainable Engineering*, 4(3):192–201, 2011.
36. T. Tielert, D. Rieger, H. Hartenstein, R. Luz, and S. Hausberger. Can V2X communication help electric vehicles save energy? In *12th International Conference on ITS Telecommunications*, pp. 232–237. IEEE, November 2012.