Quick and Energy-Efficient Routes – Computing Constrained Shortest Paths for Electric Vehicles

Sabine Storandt
FMI, Universität Stuttgart
Stuttgart, Germany
storandt@fmi.uni-stuttgart.de

ABSTRACT

In this paper we study multi-criteria routing problems related to Electric Vehicles (EVs). Because EVs still suffer from a rather small cruising range restricted by the battery's capacity, and loading stations are sparse and reloading is time intensive, previous work focused on computing the most energy-efficient routes efficiently. Unfortunately these approaches do not guarantee anything in terms of distance or travel time. But even a very eco-friendly driver might not be willing to accept a tour three times as long as the quickest one to save some energy. Therefore we present new types of queries considering energy-consumption *and* distance or travel time *and* reloading effort, e.g. computing the shortest or quickest path on which the EV does not run out of energy while requiring at most k recharging events (with k being an input parameter). The respective optimization problems are instances of the constrained shortest path problem, which is NP-hard. Nevertheless we will provide preprocessing techniques that allow for fast query answering even in large street graphs.

Categories and Subject Descriptors

G2.2 [**DISCRETE MATHEMATICS**]: Graph algorithms

General Terms

Algorithms

Keywords

Route Planning, Constrained Shortest Path, E-Mobility

1. INTRODUCTION

In recent years E-mobility has been identified as important means to reduce the consumption of fossil fuels. Electric Vehicles (EVs) are battery-powered and the necessary electricity can be produced from regenerative sources. Furthermore EVs typically exhibit lower emissions to their immediate environment in terms of combustion gases or noise levels. To accelerate the widespread transition to E-Mobility several governments offer reduced taxes for EVs compared to fuel driven cars, and provide federal funding for the development of green technologies.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACM SIGSPATIAL IWCTS'12, November 6, 2012. Redondo Beach, CA, USA
Copyright © 2012 ACM ISBN 978-1-4503-1693-4/12/11...$15.00

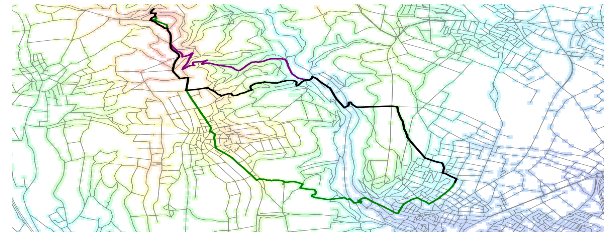


Figure 1: Small example for the diversity of paths: The energy-optimal path (green, bottom) is 9.203km long, the shortest path (purple, top) only 8.824km. The shortest path on which the EV does not run out of energy (black, middle) is only 11m longer than the shortest one.

Nevertheless Electric Vehicles still wait for their great breakthrough. One reason might be the restricted cruising range of actual EVs (up to 150km) resulting from the limited battery capacity. Indeed in use, there is the possibility to recuperate energy during deceleration phases or when going downhill (as long as the battery's capacity is not exceeded!), but still recharging will be necessary when driving longer tours. Here further inconveniences become noticeable: On the one hand loading stations (LSs) are still sparse in most countries – certainly not as widespread as gas stations – therefore one can not just plan a trip and rely on the presence of nearby LSs when the battery charge level drops. On the other hand reloading takes several hours hence it should be avoided if possible. Therefore the energy-consumption along a path should be as small as possible to stay mobile. At the same time people are certainly not willing to accept considerably longer travel times just to save a few kWh of energy. Therefore a fair trade-off needs to be achieved. The following questions reflecting *multi-criteria* objectives are natural from a user perspective (see Figure 1 for a bit of illustration):

1. *What is the shortest or quickest path on which the EV does not run out of energy?*
2. *What is the most energy-efficient path which is at most x times longer than the shortest one (e.g. $x = 1.05$)?*

A path obeying the battery constraints (i.e. the EV does not run on energy while respecting the prohibition of overcharg-

ing the battery) is also called feasible. If without reloading there exists no feasible path from source to target, the decisions how often and where to recharge have to be made reasonably. So now we have to balance distance or time to travel, energy consumption *and* reloading effort. This leads to slightly more complex optimality criteria, which are reflected in the following query types:

3. *Find the shortest/quickest feasible path with at most k recharging events.*
4. *Find a feasible path with a minimal number of recharging events and bounded distance/travel time.*

Note that the problems 2.-4. demand the input of a parameter which allows the user to modulate between eco-friendly and fast or short routes. These kind of queries are called *customizable* because the parameter is revealed at runtime only.

These four optimization problems are instances of or involve the solution of the constrained shortest path problem (CSP), which is NP-hard in general and hence the existence of polytime algorithms is doubtful. Nevertheless we will present techniques which guarantee optimal query answering in a reasonable time span for real-world networks. This enables users to plan their EV trips according to their preferences and permits navigation of EVs in large street networks.

1.1 Related Work

Because of long recharging times and the limited cruising range of EVs, the first approach for sensible route planning was finding the path with minimal total energy-consumption. Computing such an energy optimal path for an EV requires a lot more effort than conventional route planning. On the one hand the recuperation capability introduces negative edge weights to the graph, as there might be a surplus of energy on a certain road segment. This inhibits the direct appliance of Dijkstra's algorithm, which is the crucial ingredients in most route planning applications. On the other hand the EV must not run out of energy on the suggested path and is also not allowed to overcharge the battery (when recuperating). The problem of finding a path that meets these requirements (the battery constraints) while minimizing the total energy consumption from source to target was introduced in [3]. Here, the authors provided an extension of the Bellman-Ford algorithm that solves the problem in $\mathcal{O}(nm)$ (n being the number of nodes and m the number of edges). In [4] proper cost models and preprocessing techniques were introduced that reduce both the theoretical and practical runtime of a query: First the constant edge costs representing energy consumption (e.g. in kWh) were transformed into edge cost functions which fulfil the FIFO-property, hence allowing straightforward use of Bellman-Ford. Then the partly negative edge cost functions were transformed into non-negative ones by using an adaption of Johnson's shifting technique. Upon that basis Dijkstra's algorithm can be applied, leading to subsequent query times of $\mathcal{O}(n\log(n)+m)$. Finally the preprocessing technique contraction hierarchies [5] was adapted to the scenario, leading to a total speed-up of two orders of magnitude.

So very efficient retrieval of energy-optimal paths is possible now. But the energy-consumption minimizing path might not always be the desired route. Many owners of EVs can recharge their EV in their garage; therefore driving home after work, the goal is rather to get there as fast as possible than to maximize the battery load at the destination. Accordingly the quickest or shortest path that is feasible is more appropriate here. The techniques presented so far do not guarantee anything in terms of travel time or distance. Of course they also dealt with constraints, namely the battery constraints. But note that only one metric – energy consumption – was involved up to now. So the constraints and the objective were both concerned with the same metric. In contrast, incorporating travel time or distances, a second metric gets introduced to the problem and putting a constraint on the one metric while minimizing the other, we end up with instances of CSP. Therefore we can no longer hope for a polytime algorithm like for the previously studied problem. Nevertheless in [6] it was shown that speed-up techniques originally developed for the conventional shortest path problem – like contraction hierarchies or arc-flags – can be adapted to solve CSP, enabling practical query times for graphs with up to a million nodes and edges. Note that for conventional CSP instances both involved metrics are represented as constant costs along the edges. But using the energy consumption edge cost functions from [4] as one of the metrics, the setting gets more complicated and the techniques presented in [6] can not be applied straightforward.

Also reloading decisions have to be taken into account. This topic was already tackled in [7]. Here the authors aimed for a path with a minimum number of necessary recharging events. This problem can be solved in polytime because besides feasibility nothing else is guaranteed for the resulting path. The approach is reasonable under the precondition of very time intensive reloading. But currently more and more so-called battery switch stations get established. Here the complete battery system is swapped for a fully loaded one, taking considerably less time than recharging. Still having to switch the battery is an inconvenience, but the user might not accept arbitrary long detours anymore just to avoid this process.

1.2 Contribution and Outline

For the first two problems – computing the shortest feasible path and the most energy-efficient one with bounded distance – we present in Section 3 an adaption of contraction hierarchies for CSP which can also deal with edge cost functions instead of constant values only. Moreover we take loading stations into account and describe on that basis the construction of an auxiliary graph, which helps to find the optimal path efficiently. Upon that, in Section 4 we propose preprocessing techniques and algorithms, that solve the more complex optimization problems 3. and 4. where recharging is limited or minimized. Finally we evaluated the proposed approaches on real-world data. The respective experimental results can be found in Section 5.

2. PRELIMINARIES

Basically we are given the capacity of the EV's battery $M \in \mathbb{R}^+$, a street (di)graph $G(V,E)$, a cost function representing distance or travel time $d : E \to \mathbb{R}$ and a function f_e on each edge $e \in E$ representing energy consumption. These functions are modelled in consistency with [4] as follows: Let $c : E \to \mathbb{R}$ be the constant energy costs (in kWh), which might be partly negative (due to recuperation). To satisfy the battery constraints, it must be assured that no

overcharging and no running out of energy is possible. So let $e = (v, w)$ be some edge with constant cost $c(e)$, and $b(v)$ the battery load of the EV at node v. If $c(e) > 0$ the battery load $b(v)$ might be insufficient to use the edge. To make sure that no optimal path will contain e in such a case, we set the respective edge costs to ∞ if $c(e) > b(v)$. So for positive constant costs the function looks as follows:

$$f_e(b(v)) = \begin{cases} \infty & b(v) < c(e) \\ c(e) & otherwise \end{cases}$$

If $c(e) < 0$ the edge can always be used, but one must be careful not to overcharge the battery, i.e. if $b(v) - c(e) > M$ the final load at w will still be M only. This can be achieved using the following edge cost function:

$$f_e(b(v)) = \begin{cases} c(e) & b(v) < M + c(e) \\ b(v) - M & otherwise \end{cases}$$

So a feasible path $p = s, v_1, \cdots, v_k, t$ from source $s = v_0 \in V$ to target $t = v_{k+1} \in V$ is characterized by $b(v_i) \in [0, M]$ for all $i = 0, \cdots, k-1$.
In congruency with [7] we call a node w reachable from v if there exists a feasible path $p = v, \cdots, w$. The set of all nodes reachable from v is denoted by $R(v)$, the set of nodes from which v can be reached by $R^{-1}(v)$. Nodes in the latter set are also called inverse reachable from v. Both sets can be computed efficiently as described in [7].

Label Setting: In the general CSP setting, there are constant costs $c : E \to \mathbb{R}^+$ and constant resource consumptions $r : E \to \mathbb{R}^+$. Moreover on query time, we are given a source/ target $s, t \in V$ as well as a bound $B \in \mathbb{R}^+$ on the resource consumption. The goal is to find the path from s to t which minimizes the costs while not exceeding the resource bound B. One classical approach to solve instances of CSP is the label setting computation (LSC) [1]. A LSC assigns to each node v the list of all pareto-optimal tuples $(c(p), r(p))$ for an s-v-path p. Pareto-optimal means that there is no dominating path p' with $c(p') \leq c(p)$ *and* $r(p') \leq r(p)$ (and inequality holds at least once). These labels are created by an approach very similar to Dijkstra's algorithm for computing plain shortest paths. Here also labels are stored in a priority queue (PQ). A label is a triple consisting of a node ID, cost and resource consumption. The PQ sorts the labels in the increasing order of costs. The initial PQ contains only the label $(s, 0, 0)$. In every round the label (v, c, r) with minimal cost c is extracted and for the respective node v all outgoing edges $e = (v, w)$ become relaxed, i.e. if $(c' = c + c(e), r' = r + r(e))$ is pareto-optimal for w the new label (w, c', r') gets pushed into the PQ. If (c', r') dominates any solution that was already assigned to w the dominated solution gets pruned.

Simple Pruning: The number of labels that can be assigned to a single node is exponential and so is the runtime of LSC. Therefore pruning the search space is of utmost importance to achieve practical runtimes in larger graphs. A simple attempt to reduce the graph size is based on minimal resource labels [2]. Here the minimal resource label r_{min} of a node v is the minimal resource consumption of an s-t-path that visits v. Obviously, all nodes with $r_{min}(v) > B$ can never be on a feasible path, and hence these nodes as well as their adjacent edges can be excluded a priori. This condition can be checked efficiently for all nodes by running two conventional Dijkstra computations on the resource consumption starting in s, and t (on the reversed graph). Afterwards the two resulting labels $r_s(v)$ and $r_t(v)$ are summed for each node to receive $r_{min}(v)$.

Contraction Hierarchies: The speed-up technique contraction hierarchies (CH) was introduced in [5]. The basic idea is augmenting the graph with shortcuts that allow to skip a lot of edge relaxations at query time.
To that end, in a preprocessing phase an importance value is assigned to each node, and nodes are sorted in increasing order of importance. Afterwards the nodes get removed/ contracted one by one in that order, while preserving all shortest path distances in the remaining graph by inserting additional edges (so called shortcuts). More precisely, when contracting a node v for every path u, v, w the distance from u to w must stay unchanged. Therefore the edge $e = (u, w)$ has to be added if the only shortest path from u to w is u, v, w. The cost of e equals the chained costs of the edges (u, v) and (v, w). If there exists a path with costs less than the ones of u, v, w, a so called witness path is found (normally via a Dijkstra run from u to w) and the shortcut can be omitted. After all nodes have been removed, a new graph G' is created consisting of all nodes and edges of the original graph and all shortcuts. An edge $e = (v, w)$ (original or shortcut) is called upwards if the importance of v is smaller than that of w and downwards otherwise. In G' s-t-queries can be answered bidirectionally, with the forward run (starting at s) considering only upward edges and the backward run (starting at t) considering exclusively downward edges. Therefore this strategy prunes the respective search spaces dramatically. If unidirectional query answering is necessary, all edges on downward paths ending in t can be marked via a run of breadth-first-search and the forward run from s considering edges on upwards paths and the marked edges will provide the optimal solution.

In [4] the technique was adapted to the energy consumption functions. Here one must assure that all energy-optimal paths are maintained during the contraction process. In particular this means, that for every possible battery load $b(u) \in [0, M]$ the optimal path to w is not allowed to visit v in order to omit the shortcut. A CH-variant for CSP was introduced in [6]. In this scenario a witness path must dominate the reference path u, v, w in order to make the shortcut (u, w) superfluous.
In the following we will describe how these two approaches can be plugged together to solve our CSP instances with energy consumption functions as costs or resource consumptions.

3. SHORT & ENERGY-EFFICIENT PATHS

Energy-optimal Paths with Bounded Distance: Even very eco-friendly EV owners might not accept driving a route three times as long as the respective shortest path in order to save energy. But most likely they would be willing to drive a detour of e.g. 10% of the shortest path distance. Hence in a realistic setting we search for a route which obeys the battery constraints, but is not longer than some given distance bound B.
So we are confronted with an instance of CSP, where the costs are given by a function representing the energy con-

sumption, the resource usage is equivalent to the edge length, and the resource bound is given by $B = x \cdot \pi(s,t)$ with $x \geq 1$ being chosen by the user and $\pi(s,t)$ being the shortest path distance. Hence a LSC will assign tuples $(c(p), r(p) = d(p))$ to the nodes consisting of summed costs and distances. Note that the summed costs equal the difference between initial and actual battery charge status, therefore we could also assign tuples $(b, d(p))$, where the summed costs are replaced with the actual battery load. Here of course the tuple dominates $(b', d'(p))$, if $b \geq b'$ and $d(p) \leq d'(p)$.
Simple pruning can be performed conventionally in this case by running two Dijkstra computations starting at s and t respectively and considering only the distance values.
To speed up the LSC via CH, we first have to clarify how a witness is characterized in this scenario. Of course we can only omit a shortcut (u,w) for a path $p = u, v, w$ if for *any* initial battery load $b(u) = I \in [0, M]$ there exists an alternative path which is not longer than p and leads to at least the same final battery load in w as p. Observe that equivalent to the CH construction for unconstrained energy-optimal paths, there is no necessity that this is *the same path* for every I. Therefore a witness can be described as set of paths $q_1, \cdots, q_t$ with $\min(f_{q_1}, \cdots, f_{q_k}) \leq f_p$ (f being the chained energy consumption function of all path edges) and $d(q_i) \leq d(p)\ \forall i = 1, \cdots, k$. To perform the witness search efficiently we choose a set of initial battery loads $I_1, \cdots$ like in [4] but start a LSC with resource bound $d(p)$ instead of a simple Dijkstra computation for every $b(u) = I$.

Shortest/Quickest Feasible Paths: If the objective is reaching the destination as fast as possible without running out of energy, no additional parameter is required to determine the resource bound. Instead the bound is always given implicitly by the battery capacity M. In this scenario the costs are constant values (distance or travel time) while the resource consumption is described by the energy consumption function. Accordingly the simple pruning technique has to be adapted. Here we first compute $R(s)$ and $R^{-1}(t)$ and store the respective battery loads b and b_{min} for all contained nodes. All nodes $v \in V$ with $b(v) < b_{min}(v)$ are not part of any feasible path from s to t and hence can be ignored completely (this is especially true for $v \notin R(s) \cap R^{-1}(t)$).
Observe that the CH-graph can be obtained just like described in the last section (or reused), because the definition for a witness proposed there is still valid in this scenario.

Regarding Loading Stations: The presence of loading stations might affect the shortest feasible route or might be the basis for the existence of a feasible path at all. A LS is a node $v \in V$ at which the EV completely recharges whenever it visits. The set of LSs is denoted by $L \subseteq V$. To incorporate LSs efficiently, we make the following observations: Firstly an optimal path visits at most $|L|$ (all) LSs once, because any cycle starting and ending at a LS would increase the distance, but could not lead to a higher battery charge status at that vertex. Secondly the pairwise shortest feasible paths between LSs are not influenced by the choice of s and t; therefore those paths can be preprocessed. These two observations give rise to the construction of an auxiliary graph similar to the one proposed in [7]. Here we create a node for every loading station $l \in L$ and edges (l, l'), if there exists a feasible path from l to l' (without the usage of any further LSs). In contrast to the approach described in [7], we now weigh the edges with the shortest feasible path distance (we refer to this graph also as reach graph). At query time we have to add the nodes s and t and proper edges. For this purpose we first compute $R(s)$ and $R^{-1}(t)$ and intersect these sets with L to derive the directly (inverse) reachable LSs. For every $l \in R(s) \cap L$ we have to compute the shortest feasible path from s to l. Of course this can be achieved simultaneously for all of the concerned LSs via a single LSC. The same yields for t reversely. After adding the edges $(s, l), l \in R(s) \cap L$ and $(l, t), l \in R^{-1}(t)$ to the auxiliary graph, a single Dijkstra run from v_s to v_t returns the desired sequence of LSs on the shortest feasible path from s to t.

To benefit from applying CH when computing $R(s)$ and $R^{-1}(t)$ we leave all LS nodes uncontracted. Therefore they all lay on upward paths from s and downward paths to t, i.e. exploring the respective graphs is sufficient to determine the set of (inverse) reachable LSs. Moreover this speeds up the computation of the auxiliary graph, because the shortest feasible distances from one LS to all the others can be computed in the remaining (much smaller) graph with a plain LSC.

Naturally such a route might contain a large number of recharging events, because a fully loaded battery might allow for driving a much shorter/quicker tour than a heavily discharged one. In case of minimizing travel time, we can find a more reasonable path by considering also the time needed for the recharging event. These 'penalty times' have to be added to all edges in the auxiliary graph which end in a LS. Then again a single run of Dijkstra's algorithm leads to the optimal sequence of LSs, see Figure 2 for a small example.

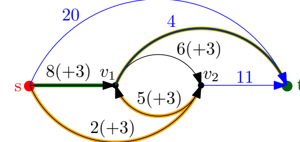


Figure 2: Computing the quickest feasible path in the presence of loading stations. If no penalty times for visiting a LS are assigned, the optimal path (orange, $c = 11$) includes two recharging events. If penalty times are added (values in brackets, here 3 units for each LS), the new optimal path (green, $c = 12 + 3$) requires recharging only once.

4. CONTROLLING THE NUMBER OF RECHARGING EVENTS

In the last section we showed how one can incorporate penalty times for recharging events in order to keep their number small on quickest feasible paths. But it is difficult to estimate the penalty times reasonably, as they actually include possibly waiting times, time to install the new battery (if it is completely switched, otherwise time to recharge), time to pay and so on. Moreover if distance is minimized instead of travel time, the optimal path still might include a lot of recharging events. Therefore we will now present two ways to control the number of recharging events without accepting long detours.

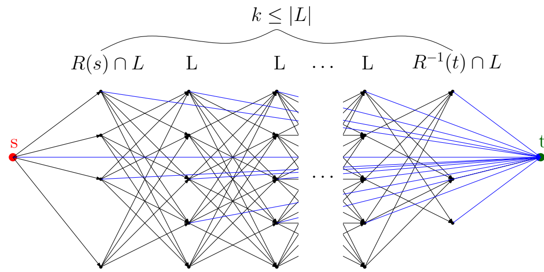


Figure 3: Layered graph to find the shortest feasible path from s to t visiting with at most k loading stations. Each node has an edge to every node in the next layer (black) – apart from edges that indicate the visit of the same LS consecutively. Nodes in $R^{-1}(t) \cap L$ have an additional edge to t (blue). Edges are weighted according to the shortest feasible path distance in G neglecting LSs (∞ if no feasible path exists).

Limiting Recharging Events: While the natural limit of recharging events is $|L|$, a driver is normally only willing to accept a few recharging events, e.g. one for small tours and three for longer tours. Therefore we would like to compute the shortest feasible path that requires at most k recharging events. For that purpose we turn the auxiliary reach-graph proposed in the last section into a *layered graph* as depicted in Figure 3. The main and invariant part of the graph consists of $|L|$ layers each containing a vertex v_l for every loading station $l \in L$. We call $i(v_l) \in \{1, \cdots, |L|\}$ the layer index of v_l. Between sequent layers j and $j+1$, there exist all edges $(v_l, v_{l'})$ $l \neq l'$ with $i(v_l) = j$ and $i(v_{l'}) = j+1$. Edges are weighted according to the shortest feasible path distance that can be achieved from l to l' in G without using any other LS. To answer an s-t-query we have to augment the layered graph as well. We insert v_s into the layered graph at layer 0 and connect it to the first layer by adding edges $(v_s, v_l)\forall l \in R(s) \cap L, i(v_l) = 1$ with the precomputed distances. Then we insert v_t at layer $|L| + 1$. To enable paths with an arbitrary number of LSs, we have to connect *every* layer to t. Hence we add edges (v_l, v_t) to the layered graph with $l \in R^{-1}(t)$ and $i(v_l) = 1, \cdots, |L|$. For a given bound k on the number of recharging events, we only connect the layers $1, \cdots, k$ to t or ignore all layers with an index of $k + 1$ or higher. In the resulting graph a plain Dijkstra computation reveals the optimal path.

Minimizing the Number of Recharging Events under a Distance Constraint: A user could also be interested to input a limit on the path length and ask for the path with a minimum number of recharging events and the length not exceeding the given limit. For this purpose we could use the same layered graph as for finding the shortest feasible path regarding LSs, but now let the distances be the resource consumption and the actual costs are 1 for all edges not adjacent to t and 0 otherwise, see Figure 4 (left) for a small example. A LSC from s to t in the layered graph with the given distance limit answers the query. But as the layered graph is a DAG (directed and acyclic) and the resource consumption for all nodes except t is equivalent to the node's layer index, the LSC will assign only one pareto-optimal tuple to every node except t, namely the index combined with the shortest path distance from s to the node in the layered graph. Accordingly there will be at most $|L| + 1$ tuples assigned to t as there are only $|L| + 1$ different cost values possible. The labels for $v \in V \setminus \{t\}$ could also be obtained by computing the shortest path distances for all nodes in the original layered graph without considering edges adjacent to t. As this graph is also a DAG the distance labels are computed layerwise. For the first layer this requires time $\mathcal{O}(|L|)$ as there are only $|L|$ incoming edges, for every subsequent layer we need time $\mathcal{O}(|L|^2)$. As soon as the distance label of a node v is settled, we relax the edge (v, t) (if it exists). If we assign a distance label to t that does not exceed the limit, we are done and the minimal number of recharging events k_{min} equals the layer index $i(v)$. Accordingly the runtime of this approach is in $\mathcal{O}(k_{min}|L|^2)$ (excluding the runtime for creating the graph with edges from/to s/t). Of course, we could also retrieve the set of all pareto-optimal solutions for t by storing the tuples $(i(v), d(v) + c(v, t))$ in a list and pruning dominated ones (see Figure 4, right). Because the nodes get parsed in layer order we only have to compare a new tuple to the tail of the list to decide whether one of them needs to be pruned. Hence we need $|L|$ comparisons and the overall runtime is $\mathcal{O}(|L|^3)$.

Observe that if for all nodes v in layer $j \geq 1$ the tuple $(j, d(v)+c(v, t))$ is not pareto-optimal for t, then for all nodes w in a layer with an index $j' \geq j$ the tuple $(j', d(w)+c(w, t))$ is not pareto-optimal for t as well. Accordingly we can abort the search for pareto-optimal labels for t as soon as we derive no pareto-optimal solution from a certain layer. While this does of course not change the theoretical runtime, it might save a lot of time in practice.

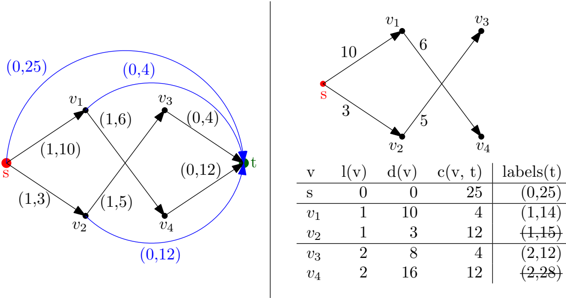
5. EXPERIMENTAL RESULTS

We implemented the described approaches in C++ and evaluated their performance on three test graphs, considering varying cruising ranges (CR) and number of loading stations (where taken into account). The test graphs are cutouts of the street network of Germany and based on OpenStreetMap data: Winnenden (W) with 500011 nodes and 1074458 edges, Baden-Württemberg (B) with 999591 nodes and 2131490 edges, and Southern Germany (S) with 5588146 nodes and 11711088 edges. All timings (t) were taken on a single core of an Intel i3-2310M processor with 2.1 GHz and 8 GB RAM.

First we computed quickest feasible paths. Without any preprocessing the LSC takes several seconds and about 5M priority queue polls in Southern German (see the third and fouth column of Table 1).The CH only adds about 1-1.6

	CR	LSC		CH		CH-LSC	
	(km)	polls	t(ms)	t(s)	edges	polls	t(ms)
W	25	$1.7{\cdot}10^4$	13	357	$2.1{\cdot}10^6$	148	3
	250	$9.1{\cdot}10^5$	433	42	$1.9{\cdot}10^6$	1316	3
B	25	$3.7{\cdot}10^4$	15	373	$4.1{\cdot}10^6$	290	4
	250	$1.2{\cdot}10^6$	681	111	$3.9{\cdot}10^6$	1576	3
S	25	$1.2{\cdot}10^6$	932	8278	$3.1{\cdot}10^7$	435	17
	250	$5.2{\cdot}10^6$	3900	1935	$2.7{\cdot}10^7$	8157	11

Table 1: Experimental results for CH-construction and computation of quickest feasible paths (averaged over 1000 random queries).



v	l(v)	d(v)	c(v, t)	labels(t)
s	0	0	25	(0,25)
v_1	1	10	4	(1,14)
v_2	1	3	12	~~(1,15)~~
v_3	2	8	4	(2,12)
v_4	2	16	12	~~(2,28)~~

Figure 4: Example for finding the path with a minimum number of recharging events under a distance constraint. On the left the augmented layered graph is given on which a label setting computation will output the optimal path. On the right an alternative approach is presented: First the shortest path distances of all nodes $v \in \{s, v_1, v_2, v_3, v_4\}$ are computed in the diminished layered graph. Then all possible pareto-optimal labels for t are derived by parsing through the nodes in layer order and combine their layer index with the minimal distance from s to t over v. Dominated tuples get pruned (crossed out in the table).

	# LS	CR (km)	reach-graph t(s)	reach-graph edges	query t(s)	reach able	avg/max # reload
W	10	50	2	38	0.027	0.83	0.54/2
	10	125	6	72	0.069	0.98	0.00/0
B	10	50	3	26	0.038	0.54	0.41/3
	10	125	8	65	0.092	0.95	0.17/1
	100	50	22	2360	0.035	1.00	1.46/4
S	10	125	58	32	0.747	0.58	0.55/2
	10	150	88	34	0.963	0.80	0.30/2
	100	50	69	679	0.128	0.58	3.20/9
	100	125	618	3114	0.834	1.00	1.12/5
	100	150	985	4528	1.150	1.00	0.86/4
	1000	100	3101	249785	0.539	0.96	1.54/5

Table 2: Experimental results for computing quickest feasible paths regarding loading stations (averaged over 10 randomly chosen sets of LS and 100 subsequent random queries each). 'reachable' denotes the fraction of targets to which a feasible path from the source (via LSs) existed.

times the number of original edges as shortcuts to the graph (see the total number of original + shortcut edges in column 6) which is comparable to the conventional case for shortest paths. Note that a small cruising range leads to more shortcut insertions, because a greater diversity of paths is optimal for different battery charge levels. Therefore – and because the runtime for pure LSC correlates with the CR – the speed-up for the 25km range is only about 4-55, while for 250km we get up to a factor of 354. Incorporating LSs increases the runtime, because more of the cruising range has to be explored to find all (inverse) reachable LS from s (from t) and the Dijkstra run on the auxiliary (reach-)graph takes additional time, see Table 2. Nevertheless we still end up with practical runtimes about a second on maximum.

Moreover we observed that the average number of reloading events seems to be acceptable for the considered scenarios, while the maximum number indicates that our approach also outputs some unreasonable routes (see the last column of Table 2). Adding reloading penalties reduces both values of course, but choosing them in a realistic fashion there a still optimal routes no user would be willing to take. Therefore we performed experiments with a limit of $k = 2$ and $k = 3$ on the number of recharging events in Southern Germany. Unsurprisingly we got comparable query times, because the layered graph is only about k times as large as the original reach-graph. Comparing the pure shortest feasible routes to the ones with the recharging limit, we saw that the ratio of reachable targets remains almost unchanged (reduction by only 4% for $k = 2$) and the average travel time (without reloading time) increases slightly by about $3 - 10\%$. So taking reloading effort into account leads to more practical routes in some cases.

Similar results were obtained for minimizing the number of recharging events under a distance constraint: Setting the bound to 1.05 times the shortest feasible path distance we could already save 1-2 recharging events on average, resulting in much more useful EV routes.

6. CONCLUDING REMARKS

In this paper we showed that the speed-up technique contraction hierarchies works for constrained shortest paths even if one of the metrics is represented by a function instead of a constant value. This allows for answering EV route planning queries, like computing the shortest path on which the EV does not run out of energy, efficiently. Moreover we proposed preprocessing approaches for incorporating reloading decisions, allowing for minimizing recharging under a distant constraint or limiting the number of such events. Partly our algorithms are customizable, i.e. the user can trade between eco-friendly and short routes with an input parameter. All our approaches guarantee optimality and were proven to work well on real-world instances. Future work includes consideration of incomplete reloading and dynamic changes in the energy consumption function, e.g. due to traffic volume, trunk load or weather conditions.

7. REFERENCES

[1] V. Aggarwal, Y. Aneja, and K. Nair. Minimal spanning tree subject to a side constraint. In *32nd ACM Symposium on Theory of Computing (STOC)*, pages 286–295, 1982.

[2] Y. P. Aneja, V. Aggarwal, and K. P. K. Nair. Shortest chain subject to side constraints. *Networks*, 13(2):295–302, 1983.

[3] Andreas Artmeier, Julian Haselmayr, Martin Leucker, and Martin Sachenbacher. The shortest path problem revisited: Optimal routing for electric vehicles. In *33rd Annual German Conference on Artificial Intelligence (KI-2010)*, 2010.

[4] Jochen Eisner, Stefan Funke, and Sabine Storandt. Optimal route planning for electric vehicles in large networks. In *AAAI*, 2011.

[5] Robert Geisberger, Peter Sanders, Dominik Schultes, and Christian Vetter. Exact routing in large road networks using contraction hierarchies. *Transportation Science*, 46(3):388–404, 2012.

[6] Sabine Storandt. Route planning for bicycles - exact constrained shortest paths made practical via contraction hierarchy. In *ICAPS*, 2012.

[7] Sabine Storandt and Stefan Funke. Cruising with a battery-powered vehicle and not getting stranded. In *AAAI*, 2012.