Journal of Artificial Intelligence Research 53 (2015) 633-658 Submitted 01/15; published 08/15

Placement of Loading Stations for Electric Vehicles: No Detours Necessary!

Stefan Funke FUNKE@FMI.UNI-STUTTGART.DE
André Nusser NUSSER@FMI.UNI-STUTTGART.DE
Universität Stuttgart
Institut für Formale Methoden der Informatik
70569 Stuttgart, Germany

Sabine Storandt STORANDT@CS.UNI-FREIBURG.DE
Albert-Ludwigs-Universität Freiburg
Institut für Informatik
79110 Freiburg, Germany

Abstract

Compared to conventional cars, electric vehicles (EVs) still suffer from considerably shorter cruising ranges. Combined with the sparsity of battery loading stations, the complete transition to E-mobility still seems a long way to go. In this paper, we consider the problem of placing as few loading stations as possible so that on any shortest path there are sufficiently many not to run out of energy. We show how to model this problem and introduce heuristics which provide close-to-optimal solutions even in large road networks.

1. Introduction

Battery-powered, electric vehicles (EVs) are an important means towards a reduction of carbon dioxide emissions when recharged using renewable energies, e.g. from solar or wind power. Despite their environmental advantages EVs still wait for their breakthrough with the main reason being their limited cruising range (often less than 200km) together with the sparsity of battery loading stations (BLSs). Planning a trip from A to B with an EV nowadays is a non-trivial undertaking; the locations of BLSs have to be taken into account, and many destinations are completely out of range.

Hence, in this early phase of E-mobility an important goal is to establish a network of BLSs so that using an EV becomes a worry-free enterprise. As modern BLSs require only little space (see Figure 1, left, for an illustration), they can be placed almost everywhere. But as this generates costs, a natural objective is to minimize the number of installed BLSs. In previous work (Storandt & Funke, 2013), a heuristic was proposed to determine BLS locations such that one can get from anywhere to anywhere in the road network without running out of energy (when choosing a suitable route). Unfortunately, this approach only guarantees connectivity but not reasonability of the routes. In fact, even rather close destinations where routes with only one recharging stop are possible, might require long detours with several recharging stops due to the placement of BLSs. A related approach by Lam, Leung and Chu (2013) suffers from similar drawbacks. In the long run, E-Mobility will only prevail if a road trip with an EV can be undertaken without unreasonable detours

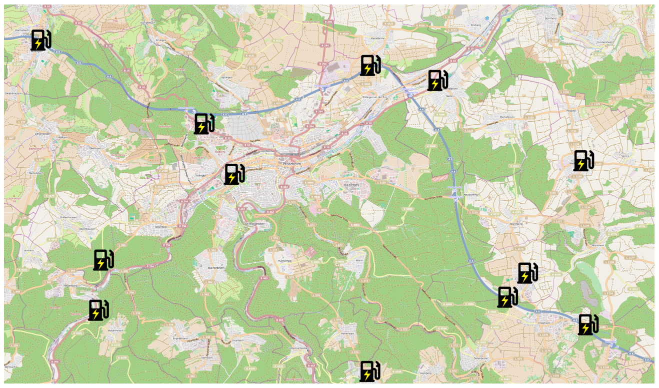

Figure 1: Inner-City battery loading station (left image), and feasible loading station cover for a small map cut-out (right image).

being introduced. In this paper we ask for a placement of the BLSs such that on *any* shortest path there are enough BLSs not to get stranded when starting with a fully loaded battery – just like it is typically the case with gas stations for conventional cars. We call such a set of BLS locations an EV Shortest Path Cover (ESC) and define the respective optimization problem as follows.

Definition 1 (EV Shortest Path Cover (ESC))**.** *Given a (di)graph $G(V, E)$, edge costs $c : E \to \mathbb{R}^+$ and a function η which for a path π decides whether this path can be traveled along without recharging the EV, the problem of determining a minimum subset $L \subseteq V$ of BLSs such that every shortest path wrt c can be traveled without running out of energy is called the EV Shortest Path Cover Problem.*

See Figure 1, right, for an idea of how a valid ESC looks like. In the remainder we will define $n = |V|$ and $m = |E|$ if not otherwise noted. Also consider that for sake of a clearer presentation we assume unique shortest paths (which can be enforced using standard techniques like symbolic perturbation). We will describe how to deal with ambiguous shortest paths towards the end of the paper. The function η captures all the energy characteristics of the network and the considered vehicle. Typically, in mountainous areas or on roads with rough surfaces, the minimal paths where an EV runs out of energy are considerably shorter than in flat terrain or downhill. For our experiments we determined the energy consumption of a road segment $e = (v, w) \in E$ with elevations $h(v)$, $h(w)$ as distance$(e) + \alpha \cdot \max(h(w) - h(v), 0)$ for some weighting parameter α (dependent on the EV). That is, the energy consumption is determined by the Euclidean distance and the height differences, similar to the energy model used in previous work (Artmeier, Haselmayr, Leucker, & Sachenbacher, 2010) but disregarding energy recuperation (negative edge costs). The function η compares for a path π the accumulated energy consumption along its edges with the EV's battery capacity $B \in \mathbb{R}^+$ to determine if recharging is necessary. Note that one could employ any kind of monotonous function η here, all approaches we will introduce in the following will work notwithstanding this particular choice.

1.1 Contribution

We describe how to model the ESC problem as an instance of the Hitting Set problem with the sets being shortest paths which require at least one battery recharge. This allows us to use algorithms developed for solving Hitting Set problems, e.g. the standard greedy approach. Unfortunately, it turns out that the difficulty of computing an ESC solution is already the instance construction. With $\Theta(n^2), n = |V|$ shortest paths in the network, extracting and storing them naively requires too much time and space to be practical. We therefore design new shortest path extraction and representation techniques, which allow to tackle even very large road networks. Moreover, we develop several refinements and heuristics which provide feasible ESC solutions more efficiently. While no a priori approximation guarantee can be shown for the solutions, we can prove a posteriori – using instance-based lower bounds – that for real-world instances the actual approximation ratio is only a small constant.

In this extended version of the original paper (Funke, Nusser, & Storandt, 2014b), we present the following new results and insights regarding ESC: We prove the ESC problem to be NP-hard (and even hard to approximate). The proven hardness is motivation and justification for the development of heuristic algorithms. Furthermore, we explain how to transform shortest paths from one (newly developed) representation into another, along with a theoretical run time analysis and experiments. The ability to transform between representations allows for more flexibility and might prove useful for other applications where compact representations of shortest paths are used. We also describe simple minimality checks for shortest paths deciding whether they need to be considered in the respective Hitting Set instance. Especially for large networks, such checks reduce the number of shortest paths that have to be stored significantly. In addition, we provide details for more involved lower bound constructions. Finally, we lay out methods for dealing with ambiguous shortest paths, for augmenting an already existing loading station set and for the case that loading station locations are restricted to a subset of the nodes in the network.

2. Theoretical Analysis

Let us first prove the ESC problem to be NP-hard, so there is no hope for efficient algorithms that solve ESC to optimality (unless P=NP). Hence in the remainder of the paper we will focus on designing algorithms that compute good approximate solutions.

Theorem 1 (Hardness)**.** *The EV-Shortest Path Cover problem (ESC) is NP-hard.*

We prove NP-hardness of ESC by a solution size preserving reduction from Vertex Cover (VC), which is one of the classical NP-hard problems. We use the following definition and notation for VC:

Definition 2 (Vertex Cover)**.** *Given a graph $G(V, E)$, the goal is to find a vertex set $C \subseteq V$ of minimal cardinality such that $\forall e \in E : e \cap C \neq \emptyset$.*

To prove Theorem 1, we show that the ability to solve the ESC problem efficiently implies the ability to solve VC efficiently as well. Hence, ESC has to be NP-hard as otherwise we would have a contradiction to the NP-hardness of VC. To that end we construct for a given

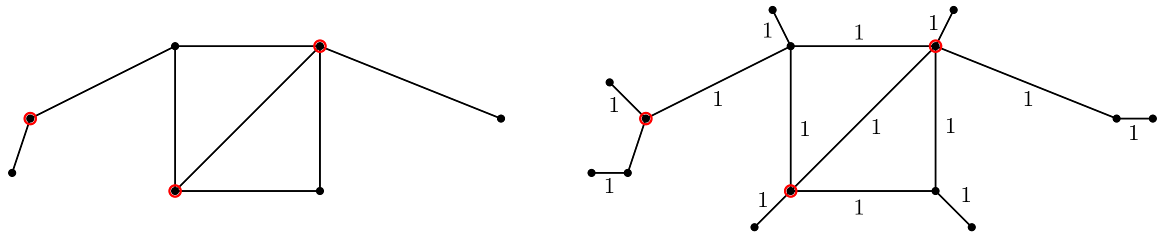

Figure 2: Left: Vertex Cover instance with optimal solution of size 3 (red circled nodes). Right: Respective ESC instance constructed by inserting an auxiliary edge per node in the graph on the left, and augmenting all edges with cost 1. The circled nodes indicate an optimal Hitting Set such that no path of cost at least 3 is unhit.

VC instance $G_{vc}(V_{vc}, E_{vc})$ a corresponding instance of ESC specified by $G(V, E), c$ and η as follows:

- $V = V_{vc}$, $E = E_{vc}$ i.e. the ESC instance initially contains all nodes and edges of the VC instance
- for all $v \in V$ add an auxiliary vertex v' and an auxiliary edge $\{v, v'\}$ to G
- for all $e \in E$ set the costs $c(e)$ uniformly to 1
- η is true for all paths consisting of less than three edges and false otherwise, i.e. every shortest path that traverses at least three edges has to have a loading station on it to make the set of loading stations a feasible ESC

The construction requires only polynomial time in the size of G_{vc}. Figure 2 illustrates the transformation from a Vertex Cover instance to an ESC instance on a small example.

We first show that any Vertex Cover solution in G_{vc} is also an ESC solution for the ESC instance constructed as described above.

Lemma 1. *A Vertex Cover C in G_{vc} yields an ESC L for G, c, η with $|L| = |C|$.*

Proof. Let $C \subseteq V_{vc}$ be a VC solution in G_{vc}. As the ESC graph G contains a corresponding vertex for every $v \in V_{vc}$, we simply set $L = C$ (so obviously $|L| = |C|$). It remains to show that after placing loading stations according to L, every shortest path in G can be traveled without running out of energy. Assume for contradiction that there exists a shortest path in G consisting of three edges $\{u, v\}\{v, w\}\{w, x\}$ and neither v nor w are in L. As every edge $e \in E_{vc}$ has at least one of its two vertices in L (because $L = C$ is a Vertex Cover for E_{vc}), it follows that $\{v, w\}$ has to be an auxiliary edge only present in G but not in G_{vc}. But auxiliary edges cannot be in the middle of a path containing three or more edges, as every auxiliary vertex has degree 1. So every path consisting of three edges contains a loading station, hence L is a valid ESC. $\square$

To complete the proof of Theorem 1, we have to show that a valid ESC solution leads to a valid Vertex Cover solution as well.

Lemma 2. *An ESC L for G, c, η yields a Vertex Cover C in G_{vc} with $|C| \leq |L|$.*

Proof. Let L be a valid ESC solution. As L might contain auxiliary vertices, we construct C by replacing every auxiliary vertex v' in L with its respective original vertex v. As v might also be part of L, we conclude that $|C| \leq |L|$. To show that C is a valid Vertex Cover in G_{vc}, we prove that for every edge $\{v, w\} \in E_{vc}$ either v or w or both are in C. Assume for contradiction that there exists an edge $\{v, w\} \in E_{vc}$ with $\{v, w\} \cap C = \emptyset$. Accordingly, neither v, nor w, nor the respective auxiliary vertices v' and w' were part of the ESC solution L. This implies a shortest path $\{v', v\}, \{v, w\}, \{w, w'\}$ of three edges with no loading station on it in G. As this contradicts L being a valid ESC solution, we conclude that for every edge we have $\{v, w\} \cap C \neq \emptyset$ and therefore C is a valid Vertex Cover. $\square$

If L is an optimal solution for ESC, then obviously $|C| = |L|$ is fulfilled in Lemma 2, as placing a loading station at an auxiliary vertex v' and the corresponding original vertex v at the same time renders the loading station at v' superfluous. Hence in an optimal solution, we never have both vertices v and v' in L. So in combination with Lemma 1, we showed that for every instance of VC, we can construct in polynomial time an instance of ESC for which an optimal solution translates into an optimal solution for the VC instance of the same size in a straightforward manner. Therefore any hardness results for VC carry over to ESC. This proves Theorem 1 and furthermore rules out the existence of $(1 + \epsilon)$ polynomial-time approximation schemes for ESC via the proven APX-hardness of a factor better than 1.3606 for VC (Dinur & Safra, 2004):

Corollary 1. *ESC cannot be approximated better than 1.3606.*

With the proven hardness, efficient algorithms that solve the ESC problem to optimality might be difficult to design unless one can make use of certain problem aspects as the battery capacity parameter or road network characteristics.

3. Modeling ESC as a Hitting Set Problem

In the following, we aim for good approximation algorithms and heuristics that solve the ESC problem in practice. In particular, we will exploit the fact that ESC can be modeled as an instance of the well-known Hitting Set problem and therefore algorithms suitable for Hitting Set computations transfer to ESC. The classical Hitting Set (HS) problem is defined as follows:

Definition 3 (Hitting Set)**.** *Given a set system $(U, \mathcal{S})$ with U being a universe of elements and $\mathcal{S}$ a collection of subsets of U, the goal is to find a minimum cardinality subset $L \subseteq U$ such that each set $S \in \mathcal{S}$ is hit by at least one element in L, i.e. $\forall S \in \mathcal{S} : L \cap S \neq \emptyset$.*

In our case, U consists of all nodes of the road network (the possible BLS locations). The set $\mathcal{S}$ is composed of the vertex sets of all shortest s-t-paths (excluding s and t themselves) for which a fully charged battery at s does not suffice to reach t. Our function η characterizes the paths where the energy consumption of traversing them exceeds the battery capacity $B \in \mathbb{R}^+$ – we call these paths *B-violating.* Clearly, we only need to consider set-minimal paths as supersets are hit automatically. Theorem 2 shows that this Hitting Set formulation indeed solves our ESC problem (see also Figure 3 for an illustration).

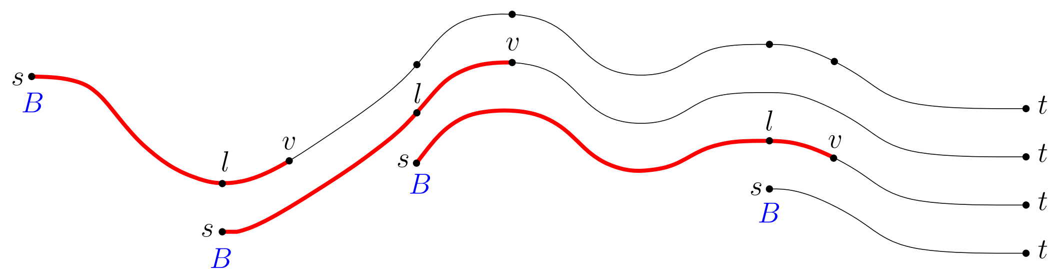

Figure 3: If a shortest path from s to t exhibits a B-violating prefix (s-v, marked red), then according to the Hitting Set formulation there has to be a loading station l on this subpath. As the vehicle fully reloads at l, the same argumentation applies to the subpath l-t, which is illustrated in the picture by cutting off the prefix s-l in every layer. The final s-t-path is not B-violating anymore. Therefore the originally considered s-t-path can be traveled when visiting the three indicated loading stations. The blue B's mark the nodes where the battery is fully loaded, i.e. where the charge level is equal to the battery capacity.

Theorem 2 (Correctness)**.** *The Hitting Set formulation leads to a feasible ESC solution, i.e. when placing loading stations according to the Hitting Set solution L, every shortest path in G can be traversed with an EV without running out of energy according to η.*

Proof. Let $\pi(s,t)$ be a shortest path from s to t in G which is B-violating, and let the EV be fully loaded at s. Let $\pi(s,v)$ be the minimal B-violating prefix of $\pi(s,t)$. This prefix has to be hit by a loading station $l \in L$ with $l \neq s, l \neq v$ as demanded by the Hitting Set formulation. The EV can reach l from s, as $\pi(s,v)$ is the minimal B-violating prefix and l appears before v on $\pi(s,v)$. At l the EV is fully re-charged. Hence the whole argumentation transfers now to the subpath $\pi(l,t)$. Applying this argument recursively, the EV will finally reach a loading station on $\pi(s,t)$ from which the suffix of the path is no longer B-violating. Therefore the EV will reach t via $\pi(s,t)$ without running out of energy. □

Note that there is no precision loss in reformulating ESC as a Hitting Set problem instance, since every solution to ESC is a feasible solution of exactly the same cardinality for the respective Hitting Set problem instance.

At this point, common Hitting Set solving techniques can be applied to solve ESC, e.g. the standard greedy algorithm. The greedy algorithm repeatedly picks the node hitting most so far unhit sets in $\mathcal{S}$ and adds it to the solution. It terminates as soon as all sets are hit. A solution computed with the greedy algorithm is guaranteed to be a $((\ln |\mathcal{S}|) + \Theta(1))$-approximation (Chvatal, 1979); we ignore the lower-order term $\Theta(1)$ from now on. In our application, the number of sets in the system is upper bounded by the number of shortest paths in the graph. Therefore, we have $|S| \leq n^2$ and hence the greedy algorithm provides a $2\ln(n)$ approximation guarantee. The running time of the greedy algorithm depends crucially on fast access to the so far unhit sets in $\mathcal{S}$ in each round.

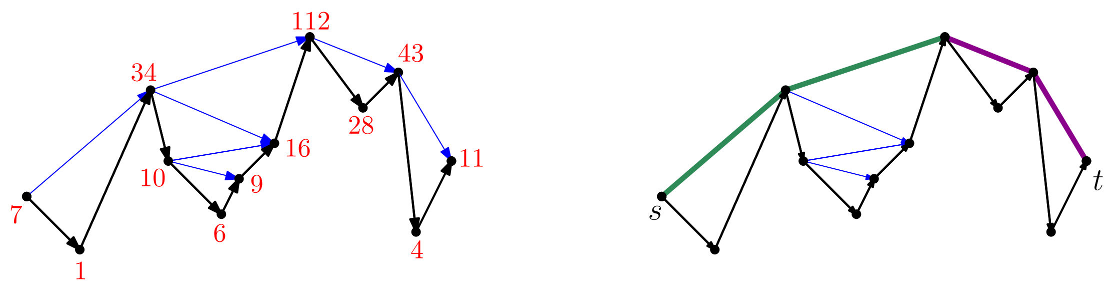

Figure 4: Cut-out of a CH-graph: Black edges are original edges, blue edges indicate shortcuts. The red node labels reflect the contraction order. So, for example, at the moment the node labeled 6 was contracted, the shortcut from node 10 to node 9 was inserted as the shortest path from 10 to 9 went over 6. On the right side, the search space for a query from the leftmost to the rightmost node is illustrated. Green edges show $G^{\uparrow}_{out}(s)$ and purple edges $G^{\downarrow}_{in}(t)$.

In the remainder of the paper, we will investigate the efficient construction of the set system using different path extraction and representation schemes and study their influence on the greedy algorithm.

4. Basics for Practical ESC Computation

To determine suitable BLS positions, we first have to construct the set of shortest paths on which the EV would run out of energy (according to η). Computing the shortest path between two nodes or from one to all other nodes is classically performed using Dijkstra's algorithm (also shortly called 'Dijkstra'). In large street networks Dijkstra is too slow to process a large number of such queries (as it is necessary for our application), though. Therefore, we will instrument speed-up techniques developed for accelerating shortest path queries to achieve better running times for our approaches. In particular, we will employ *Contraction Hierarchies (CH)* (Geisberger, Sanders, Schultes, & Delling, 2008) for this purpose. The basic idea behind CH is to augment the graph $G(V, E)$ with a set E' of so called shortcuts, which span (large) sections of shortest paths. Using these shortcuts instead of original edges allows for a dramatic reduction of operations in a Dijkstra run.

The central operation in the CH preprocessing is that of a *node contraction*. Considering a graph $G(V, E)$ with a node v to be contracted, the goal is to remove v from G without affecting shortest path distances between all remaining nodes. This can be achieved by creating additional shortcut edges between neighbors of v as follows: For every pair of neighbors u, w of v with $(u, v), (v, w) \in E$, a shortcut (u, w) is created (with cost of the path uvw) if uvw is the *only* shortest path from u to w. Then the resulting graph (with v removed and all necessary shortcuts added) exhibits the same shortest path distances as the original graph. The CH preprocessing phase instruments this node contraction by first assigning a label $l : V \to \mathbb{N}$ to each node. Then, the nodes are contracted in increasing label order. Having contracted all nodes and constructed a set of shortcuts E' on the way, we return as result of the preprocessing phase the CH-graph $G'(V, E \cup E')$, that is, the original graph augmented with the shortcuts (and the labeling), see Figure 4 (left) for an example.

According to these labels, original or shortcut edges (v, w) are referred to as upwards if $l(v) < l(w)$ and downwards otherwise; paths are called up/downwards if they consist exclusively of edges of that type. It can be shown that by the way we created shortcuts, for each node pair $s, t \in V$ a shortest path exists in $G' = G(V, E \cup E')$ which can be decomposed into an upward path starting at s followed by a downward path ending in t; The highest node of the path wrt l is called the *peak node* in the following. This property of the path allows to restrict a bidirectional Dijkstra run to $G^{\uparrow}_{out}(s)$ and $G^{\downarrow}_{in}(t)$ which refer to the subgraphs of G' containing only all upwards paths starting in s or all downwards paths ending in t respectively. In Figure 4 (right), these subgraphs are illustrated. The resulting optimal path found by the bidirectional Dijkstra has the same costs as the shortest path in the original graph. But the representation of the path is different, because the path now consists (partly) of shortcuts. To get the shortest path in the original graph, an unpacking procedure is applied. For every shortcut, the two edges (original or shortcut) it directly spans are memorized during CH construction. Thus, for unpacking one just has to recursively replace shortcuts by these spanned edges until all original edges are identified.

Note that the CH scheme can also be employed to just *represent* a shortest path π very concisely by replacing as many subpaths of π as possible by shortcuts. We also call this representation of π a *CH-path.* In our experiments it turns out that CH-paths are an extremely economic representation scheme for shortest paths.

For the one-to-all shortest path problem, the PHAST algorithm (Delling, Goldberg, Nowatzyk, & Werneck, 2011) also takes advantage of the CH-preprocessing scheme. Here in a first phase all nodes in $G^{\uparrow}_{out}(s)$ for a source $s \in V$ are settled via a Dijkstra run. In the second phase all downward edges (v, w) are relaxed in decreasing order induced by $l(w)$, thereby computing correct distances to all nodes in V. So the second phase is simply a sweep over a subset of the edges which requires only linear time. Correctness of PHAST results again from the fact that for every node t the shortest path from s to t can be decomposed into an upwards path (with all contained nodes settled in the first phase) and a downwards path. As the set of downwards edges forms a directed acyclic graph and the labels $l(w)$ induce a topological order, a sweep over the edges in this order assures that at the moment an edge (v, w) is relaxed, node v is already settled. Hence PHAST computes exact shortest path distances from s to all nodes in the network. Note that for a single shortest path query PHAST is not the method of choice, as other techniques like pure CH-Dijktra computations normally run in time clearly sublinear in the number of edges in practice.

5. Construction of the Set System

In this section, we investigate several strategies to extract the set system for a given ESC instance, i.e. the set of all minimal shortest paths in G which are B-violating. Along with different extraction strategies, we present different ways to represent and store the respective set of shortest paths and discuss the advantages and disadvantages of these representations.

5.1 Naive Extraction

The simplest approach that comes to mind is to compute the shortest path tree (via Dijkstra) for every $s \in V$ and to identify all nodes in the tree with accumulated energy cost values above B. Once all nodes in the priority queue of Dijkstra have settled predecessors that already belong to B-violating paths, we can abort the exploration from that source s. The respective paths in the search tree can then be backtracked and stored as e.g. complete vertex sets. For small exploration radii (small bounds B) this is a practical approach, but for larger B the space consumption of $\mathcal{O}(n^2\sqrt{n}), n = |V|$ is enormous (assuming an average path length of $\sqrt{n}$). Even if we only store the Dijkstra search tree for each $s \in V$ via predecessor labels, we still have a space consumption of $\mathcal{O}(n^2)$. Of course, we could easily achieve linear space consumption in the number of B-violating paths, by only storing the source vertex s and the target vertex t for each path. But if we want to have access to the nodes on a certain path, we again have to run a Dijkstra computation in the network from s to t. For huge sets of paths computing the nodes between s and t always on demand is very time-intensive. No matter which representation we use to store the paths, the time complexity of $\mathcal{O}(n \cdot (n \log n + m)), m = |E|$ for the naive extraction already limits usability for real-world instances; $\mathcal{O}(n \log n + m)$ being the runtime for one Dijkstra run.

In fact, this is also the main difficulty for other Hitting Set-type problems on street networks. For example, speed-up techniques for shortest path queries like *Transit Nodes (TN)* (Bast, Funke, & Matijevic, 2009) or *Hub Labeling (HL)* (Abraham, Delling, Goldberg, & Werneck, 2012) are based on hitting a certain set of shortest paths as well. Methods for complete instance construction are impractical there. Therefore several custom-tailored heuristics were developed that allow for efficient computation without explicitly constructing $\mathcal{S}$ (Arz, Luxen, & Sanders, 2013). But their setting differs significantly from ours, as in our setting η poses an additional criterion to c for identifying the paths contained in $\mathcal{S}$. Hence the distance bound employed in their setting leads to a set of equal length paths, while in our scenario, due to different energy consumption when driving uphill or downhill, the lengths of minimal B-violating paths differ vastly. So unfortunately these TN and HL (and other related) heuristics do not carry over to our setting. Therefore we need to explore new ways of extracting and storing shortest path sets.

5.2 PHAST-Based Extraction

For large bounds B finding all B-violating paths from a source node resembles the one-to-all shortest path problem. PHAST was explicitly designed to solve this task efficiently. The paths we can backtrack in the respective search tree are in CH-representation, i.e. they consist partly of shortcuts. This is a huge advantage compared to conventional paths in terms of storage, because with shortcuts spanning large portions of the shortest path the number of nodes in the CH-path is significantly smaller (about two orders of magnitude for the street network of Germany). There are some downsides, though: Nodes are processed in the second phase of PHAST in l-order and not increasingly by distance; hence incorporating B as stopping criterion seems difficult. Moreover, if B is not that large or leads to paths with vastly differing lengths, the $\Omega(n^2)$ lower bound on the (accumulated) runtime of PHAST from every source might already result in a large overhead. Hence we propose a different strategy which is also based on CH but has the potential of being significantly faster.

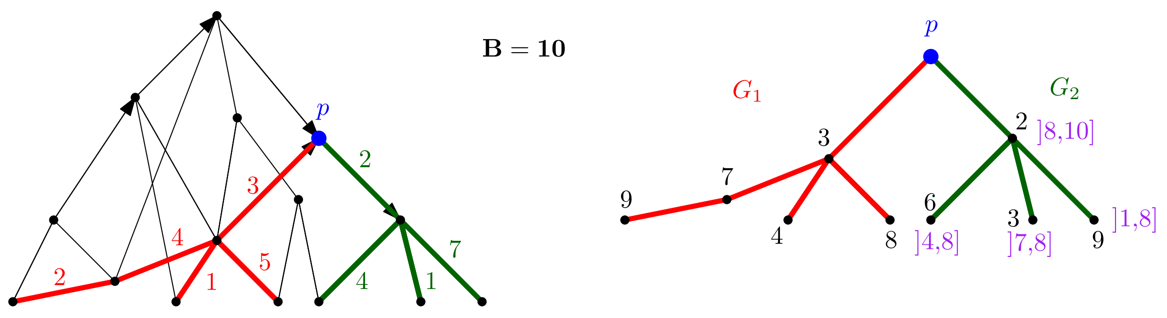

Figure 5: Left: Schematic representation of a CH-graph with the height of the nodes indicating the contraction order. For the blue-marked peak p, $G_1 := G^{\uparrow}_{in}(p)$ is colored red and $G_2 := G^{\downarrow}_{out}(p)$ is colored green. Note that for all nodes in the bottom layer, both G_1 and G_2 are empty and nothing has to be done (which in practice is true for about 50% of the nodes). Right: Energy cost labels (black) assigned to G_1 and G_2 resulting from two Dijkstra runs starting at the blue node. The resulting intervals for nodes in G_2 are expressed in purple. So if we, for example, search for matching targets for the source node labeled 3 in G_1, the intervals reveal that only the node labeled 9 in G_2 is a suitable candidate.

5.3 Peak Node Mapping (PNM)

A large number of B-violating paths can originate from a source $s \in V$. Exploring all these paths with Dijkstra or PHAST is very time-consuming. The core idea of PNM is to enumerate B-violating paths completely different but also taking the CH-representation of paths into consideration. As explained above, shortest CH-paths are unimodal with respect to the labeling l and the node with the maximal label is called the *peak*. Intuitively, nodes with a high label appear in more shortest paths as peaks. In fact, in real-world graphs, the 5% nodes with highest level constitute the peaks of all reasonably long shortest paths. This gives rise to a path enumeration algorithm, which explores paths not from the source but from the peak, resulting in dramatically reduced search spaces for the majority of nodes.

Algorithm. Our PNM algorithm works as follows: We consider one by one every node $p \in V$ as potential peak. As all shortest paths with peak p can only contain further nodes with a smaller label, we only need to search upwards paths ending in p and downwards paths starting in p for prefix and suffix candidates. The respective subgraphs of the CH-graph G' containing these paths are called $G_1 := G^{\uparrow}_{in}(p)$ and $G_2 := G^{\downarrow}_{out}(p)$, see Figure 5, left, for an illustration. A conventional Dijkstra run in each of G_1 and G_2 (which are typically very sparse) reveals the distances from p to the contained nodes. Now we are interested in combinations of shortest (upward) paths in G_1 and shortest (downward) paths in G_2 leading to minimal B-violating paths. Testing them all naively is too expensive. Therefore, we construct for each p an *interval tree* (Berg, Cheong, Kreveld, & Overmars, 2008) on the nodes in G_2. The interval $]a, b]$, which we associate with such a node t, denotes the range of possible energy consumption values of a path prefix $\pi(s, p)$ in G_1 such that $\pi(s, p) \cup \pi(p, t)$ is a *minimal* B-violating path. These intervals can easily be computed by a single pass over the Dijkstra search tree in G_2, see Figure 5, right, for an example. So for every possible

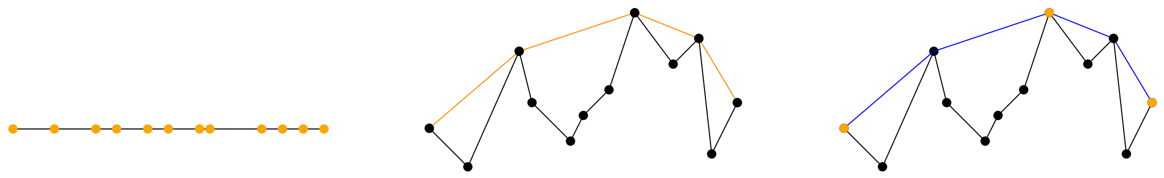

Figure 6: Illustration of three methods for representing and storing shortest paths (stored elements are always colored orange): On the left as complete vertex set, in the middle as shortcut set/CH-path (the heights of the vertices in the image correspond to their l-value), and on the right as triple consisting of source, target and peak vertex.

source $s \in G_1$, we query the interval tree for the set of targets T in time $\mathcal{O}(\log(|G_2|) + |T|)$ storing the resulting paths as quadruples $(s, p, t, c(s, p) + c(p, t)), t \in T$. Note that for the employment of the interval trees we make use of the special choice of η. For different choices of η the interval computation procedure has to be adapted.

Filtering. After all nodes are processed, we have a set of B-violating paths from which, unfortunately, not all are shortest paths. The concatenation of two shortest paths ($\pi(s, p)$ and $\pi(p, t)$) does not need to be a shortest path itself. So it remains to filter this set appropriately. This can be achieved by using distance oracles with quasi constant look-up time as e.g. provided by HL or by another pass over all nodes in the role of the peak, always pruning a quadruple if for s, t a shorter path was found for $p' \neq p$. Note, that pruning can already be employed during the construction phase if the intermediate path set sizes become too large. The final set of paths is then stored as list of triples (s, p, t) – an even more compact representation than CH-representation. In Figure 6, a visual comparison is provided for storing a path as vertex set, as shortcut set, or as PNM triple.

Accessing all nodes in the respective shortest s-t-path in G as required for the greedy Hitting Set algorithm is no longer trivial for our sophisticated representation schemes, though. Therefore, we develop suitable adaptions of the greedy algorithm to work on the CH-representations and on PNM triples in Section 6.

5.4 Minimality Checks

As mentioned when introducing the Hitting Set formulation of ESC, we only need to extract and store *minimal* B-violating shortest paths, i.e. paths with no subpath being B-violating as well. Adding non-minimal paths to the set system of course does not invalidate the solution, but increases the complexity of storing all sets, and the running time of the greedy algorithm later on.

For the naive extraction scheme, a B-violating path $\pi = sv_1 \dots v_k t$ identified in a Dijkstra run from s might not be minimal, as the path might still be B-violating without some prefix. More precisely, we have to check if the path remains B-violating when removing the first edge (s, v_1), and if so, we don't have to include π into $\mathcal{S}$. The respective Dijkstra run when considering v_1 as source will ensure that no B-violating path is missed.

For the PHAST-based extraction, the minimality check becomes more complex. As the paths are in CH-representation, we might not have direct access to the first edge of the path in the original graph (if the path starts with a shortcut). So we have to unpack the first shortcut before performing the minimality check.

For PNM both prefix and suffix deletion might lead to a subpath that is still B-violating. So immediately after the decision in favor of a triple (s, p, t), we check for the first edge on the path from s to p and the last edge on the path from p to t if their removal will not destroy the property of the path being B-violating. Again, like with PHAST, paths are in CH-representation during the construction, therefore we have to unpack the shortcuts first.

5.5 Transformability

Note, that the extraction scheme does not tie us to a certain path representation. In fact, all mentioned representations (vertex sets, source-target-pairs, CH-paths, triples) can be converted into each other. Especially the transformation from vertex sets to CH-paths will turn out to be favorable, as CH-paths yield a fair trade-off between space consumption and applicability of the greedy algorithm as explained in more detail in Section 6. We will now provide the details for all transformations including theoretical transformation times. For the latter we assume that the complete vertex set representation contains k elements.

- *From source-target-pairs to vertex sets.* Given s, t, the complete path is computed via a Dijkstra run in G and backtracking, requiring $\mathcal{O}(n \log n + m)$ time.

- *From vertex sets to CH-paths.* We assume the CH-labels $l : V \to \mathbb{N}$ are available. Then a recursive procedure allows to turn the vertex set into a CH-path. We first identify the node v_0 in the vertex set with the highest l-value (the peak). Then we split the vertex set into the prefix before v_0 and the suffix after v_0. For both of those sub-paths we again search for the node with the highest l-value, providing us with v_1, v_2. These two nodes are connected with v_0 via a direct edge/shortcut in the CH-graph (as all nodes in between were contracted before), so we have $(v_1, v_0), (v_0, v_2)$. This again provides us with a prefix (nodes before v_1) and a suffix (nodes after v_2) on which we recurse. The algorithm stops when the prefix is only s and the suffix only t (or the prefix is monotonously increasing wrt l and the suffix monotonously decreasing), see Figure 7, top, for an illustration. Assuming the CH-representation contains h shortcuts, the transformation can be performed in $\mathcal{O}(k \cdot h)$.

- *From CH-paths to PNM triples.* We just need to extract the peak vertex (besides source and target) which can be done in $\mathcal{O}(h)$ if the CH-path consists of h shortcuts.

- *From PNM triples to CH-paths.* Given source s, target t, and peak p, we run a Dijkstra starting at p in $G_1 := G^{\uparrow}_{in}(p)$ and $G_2 := G^{\downarrow}_{out}(p)$ until s and t are settled. As the number of edges in a CH-graph is assumed to be in $\mathcal{O}(m)$, the runtime for a single transformation is in $\mathcal{O}(n \log n + m)$ just like for the transformation from source-target-pairs to vertex sets. But as typically a peak with a small l-value has very small G_1 and G_2, and a peak with high l-value generates many s-t-paths at once, the amortized costs per path are considerably smaller.

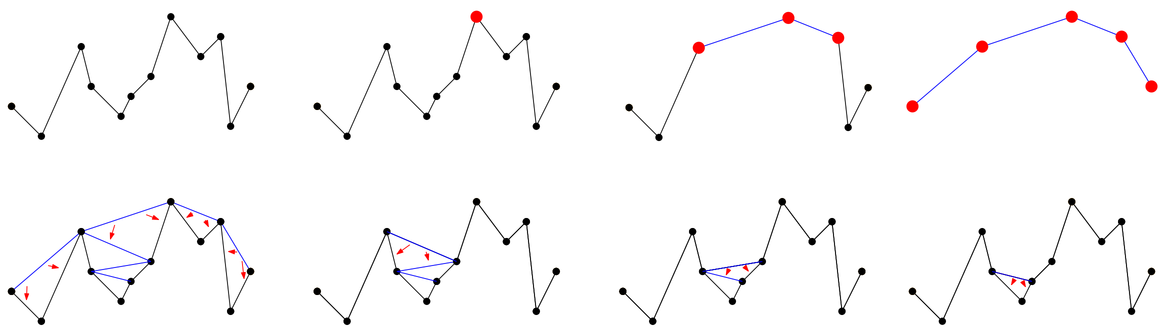

Figure 7: From vertex sets to CH-paths and back. In the first row, a vertex set is given. The l-values derived in the CH construction are indicated by the vertex elevations. Then recursively the vertex with the highest l value (marked red in the images) in the prefix and suffix of the path is extracted and shortcuts are added to span path sections with lower l-value. In the second row, a CH-path is recursively unpacked. Shortcuts are colored blue. The red arrows point to their replacement edges in the next unpacking step. The final path is the same we started from in the first row, and does not contain any shortcuts.

- *From CH-paths to vertex sets.* Given a CH-path, we apply the unpacking method described in Section 4, i.e. we recursively replace shortcuts by their spanned edges until the path consists of original edges only (see Figure 7, bottom). If the resulting path consists of k vertices, the unpacking can be performed in $\mathcal{O}(k)$.
- *From vertex sets to source-target-pairs.* The first and the last vertex of the complete path are stored and the rest is neglected. This transformation costs $\mathcal{O}(1)$ or $\mathcal{O}(k)$ if we consider the deletion of the $k-2$ other elements as well.

In the following, we will no longer investigate the source-target pair representation, as storing PNM triples requires only one item more per path but at the same time allows for more efficient access to the path's vertices.

6. Greedy Hitting Set Computation

As explained above, the greedy approach is a natural strategy to solve the Hitting Set problem approximately. Theoretically it yields solutions within a factor of $2\ln(n)$ of the optimum in our setting. But in practice greedy performs much better than the theoretical, *a priori* approximation guarantee implies.

For all but the simplest set system representation the application of the greedy Hitting Set algorithm is not straightforward and requires some deliberate operations on the set system/path representations as we will see in the following.

6.1 Complete Vertex Sets

If the paths are simply given as the set of contained vertices, a single scan over all these sets can determine the 'best' node hitting most paths. Another scan can remove the paths that have been hit by the selected node. These two scans can also be combined into one by updating the counter values when removing the newly hit paths (an initial count is still necessary). Unfortunately, the space consumption of this approach is enormous, also making a single scan quite expensive.

6.2 CH-Paths

When representing the minimal B-violating paths as CH-paths, we could convert the single paths into original paths by unpacking the shortcuts and then operate on the complete vertex set of the path. Note that the paths would be processed one by one and only one unpacked path would be kept in memory at a time. But there is a much better strategy than just uncompressing every single CH-path to get to the original node sets: maintaining a usage counter for each edge (counting how many shortest paths use that edge), we first scan over all edges of all CH-paths in the set system to be hit, incrementing the respective counters. Then we traverse all shortcut edges of the graph in decreasing order of their construction in the CH-preprocessing, incrementing the counters of the spanned edges. The node counters, maintained to identify the maximum node, can then be derived by a final scan over all original (non-shortcut) edges. Keeping reverse information about which edges are spanned by which shortcut also allows the identification of all sets that have been hit by a node. If we update the edge counters when removing CH-paths from the set, picking a node requires only one scan over the edges to push the counts down on the non-shortcut edges, one scan over the usage counters and one scan over the set of CH-paths.

6.3 Peak Node Triples

When paths are described as triples of source, target, and peak node, we can get the CH-path representation as described in Section 5.5. Then we proceed as with the CH-path representation. Note, that this CH-path representation is computed on demand for each peak in every round to avoid keeping all paths in CH-representation permanently in memory.

7. Multi-stage Construction

For country-sized graphs, even the improved set system extraction methods and representations do not reduce the space and time consumption enough to be practical. Therefore we introduce a procedure which interleaves the set extraction and the greedy Hitting Set computation in a multi-stage algorithm. This multi-stage algorithm requires significantly less space than the complete construction of the set system, and therefore can be applied to considerably larger instances.

7.1 Nested Hitting Sets

For an instance of our ESC problem determined by the battery capacity B, we make the following important observation: For every capacity $B' \leq B$, a Hitting Set L' for the instance corresponding to B' is also feasible for the original instance (having enough BLSs for a smaller battery capacity also suffices for a larger battery capacity). So, for example, if $B = 20$kWh, solving the problem for e.g. $B' = 5$kWh would be feasible as well. While the construction of L' for some $B' \ll B$ might be considerably faster due to smaller exploration

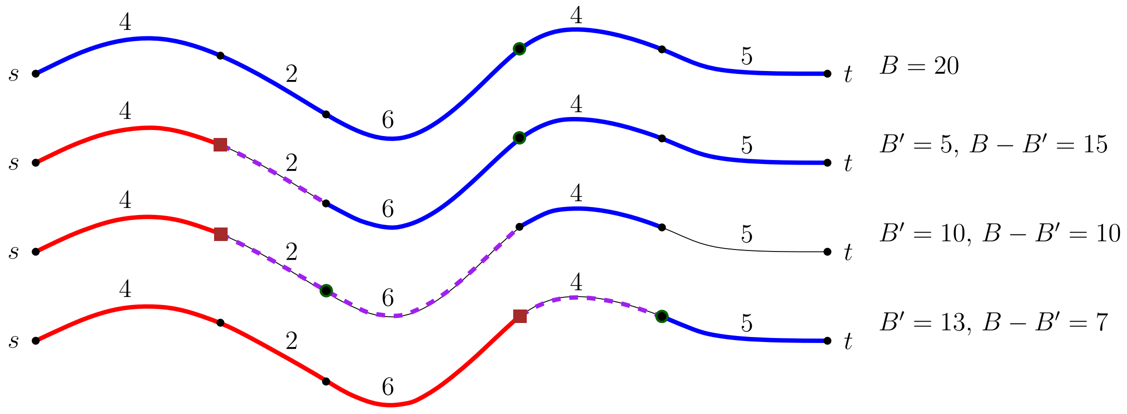

Figure 8: If the battery capacity is $B = 20$ kWh, the shortest path from s to t needs to be hit by a loading station as it exhibits energy costs of 21. In the conventional construction, the complete s-t-path would be part of the Hitting Set instance. The three lower images illustrate what happens when we use a nested construction with different values of B'. The red+purple/dashed subpath indicates the minimal B'-violating path starting at s. The brown square is a possible hitter for this path. The purple/dashed+blue path indicates the minimal (B-B')-violating path starting at the brown square. This path is always a subpath of s-t no matter how B' is chosen. So hitting the blue+purple path (large dot in the image) assures that all B-violating paths are hit as well. The nodes marked by brown squares are not part of the final solution.

radii, L' is typically also much larger than necessary for the instance defined by B. So simply using the solution for $B' = 5$kWh although the real battery capacity is $B = 20$kWh, we expect a result with many superfluous loading stations.

But there is another advantage of first quickly computing a Hitting Set for a small value B': It allows us to construct a new, smaller problem instance for which any feasible Hitting Set L'' is also feasible for our original problem (defined by B) – and L'' is hopefully much smaller than L'. This second instance is defined by the set of paths originating in L' that are (B-B')-violating.

We prove in Lemma 3 that the Hitting Set L'' for the second instance is indeed also a Hitting Set for the original instance.

Lemma 3. *Given a battery capacity B, and a second capacity $B' < B$. Let L' be a feasible ESC solution for B', and L be a feasible Hitting Set for all minimal shortest (B-B')-violating paths originating in L', then L is a valid ESC solution for the battery capacity B.*

Proof. Consider some B-violating s-t-path π in the original instance. Then π must be hit by a node $v \in L'$ less than B' away from s. The path $v, \ldots, t$ from this hitter to the target (or a prefix thereof) has to be in the new constructed path set of (B-B')-violating paths originating in L'. Therefore this subpath has a hitter in L. Hence L hits any B-violating shortest path, which makes L a valid ESC solution for battery capacity B. Figure 8 illustrates the proof on a small example. $\square$

7.2 Path Cover

For very small values of B, we can even compute Hitting Sets without any exploration and evaluation of the η function purely based on the connectivity structure of the graph using a so-called *k-Hop Path Cover* (Funke, Nusser, & Storandt, 2014a) which is a generalization of a *Vertex Cover*. We construct a set of vertices $C \subseteq V$ such that *any* directed (not necessarily shortest) k-hop path in G contains at least one vertex from C (for $k = 1$ this is simply a Vertex Cover). Then C is an ESC solution for B^* where B^* is the maximal energy cost of a k-hop path, which can easily be upper bounded by k times the maximal energy cost of an edge. For values $k \leq 48$ this takes only a few minutes even on large graphs using a variant of depth first search, making this step negligible for the overall running time.

7.3 Combination

In our implementation we combined nested Hitting Sets and k-Hop Path Covers to a multi-stage procedure, constructing a sequence of Hitting Sets $L_r, L_{r-1}, \cdots, L_1 = L$ for a sequence of values $B_r < B_{r-1} < \cdots < B_1 = B$, finally returning L as the Hitting Set for the given instance.

The first B_r results from a k-Hop Cover with small value of k, for all subsequent solutions we apply the nested Hitting Set approach (and choose B_i manually). There might be some loss in terms of quality compared to the greedy algorithm on the full set system due to the nested construction. Our experimental evaluation will show, though, that the loss in terms of quality is not that pronounced, but the running times are drastically improved, and the graph sizes which we can handle with this approach are much larger.

8. Refinements and Lower Bounds

In this section we introduce a speed-up strategy for the greedy algorithm which is independent of the employed set representation. Then we develop algorithms to construct instance-based lower bounds for the ESC solution. These bounds will be helpful in our experimental evaluation, as they prove – a posteriori, after running our algorithms – that the computed solutions are in fact pretty close to the optima.

8.1 Multiple Hitters Heuristic

Even with the non-naive representations, there is considerable work involved when picking the next 'best' node in the greedy algorithm. So it might be worthwhile to add several nodes to the Hitting Set in each round. Normally, we pick only the node which hits most so far unhit sets, and refrain from picking other nodes in the same round as picking the first node influences the hit counters of others. On the other hand, if we pick nodes that do not interfere with each other, the quality of the solution should not decline too severely. One way to achieve this is to generate the list of nodes sorted in ascending order of their hit counters, always picking the first one and then going down the list selecting the next nodes which have the shortest path distances of at least D to all nodes already picked. Here D is appropriately chosen, e.g. an upper bound on the longest shortest path that is not B-violating. Thereby we make sure no path in our set increased the hit counter of two or more nodes picked in one round.

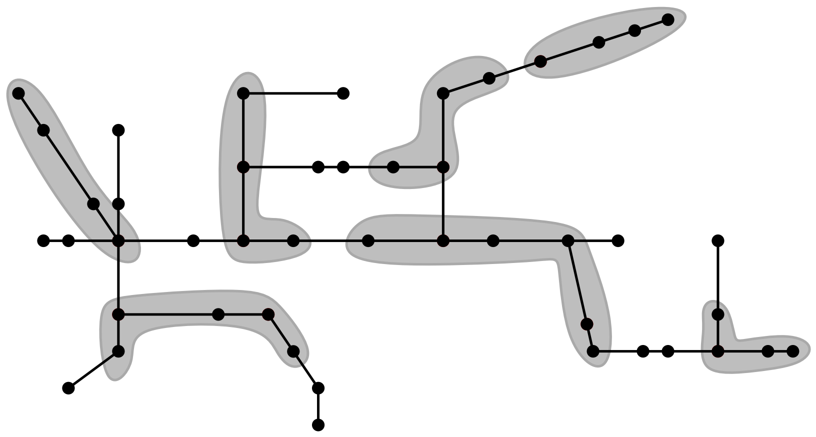

Figure 9: Set of seven node-disjoint B-violating paths (highlighted in grey) in a small example graph. Every valid Hitting Set for all B-violating paths in the graph has to contain at least seven vertices, as no vertex can hit more than one of the grey marked paths. Therefore the size of a set of node-disjoint paths is a feasible lower bound for the optimal Hitting Set size.

8.2 Simple Instance-Based Lower Bounds

To evaluate the quality of our heuristics, we would like to compare the outcome to the optimal solution. But as the optimal value is typically unknown, we instead compare to a good, but easily computable lower bound. In a study on Transit Nodes (Eisner & Funke, 2012) a rather involved lower bound was proposed, which takes an effort comparable to solving the Hitting Set problem itself. We propose a much simpler alternative which suffices for our purposes: As a by-product of the generation of the set system itself we can obtain a *set of node-disjoint B-violating paths*. So no two paths in the set have a non-empty intersection. Clearly, any feasible solution must contain an extra node per path in this set. Hence the size of a set of such node-disjoint paths yields a valid lower bound, see Figure 9 for an illustration.

In case we do not generate the set system explicitly (because we use nested Hitting Sets), we can greedily extract a set of node-disjoint paths by running Dijkstra computations from random sources and adding B-violating paths to our set as long as they do not intersect with previously selected ones. The size of the set provides a valid lower bound at any time.

9. Dealing with Real-world Settings

Throughout the paper, we made some assumptions about the ESC problem for sake of a clean definition and easier algorithm descriptions. As these assumptions are not necessarily met in practice, we explain in the following how to adapt our algorithms to still perform well under real-world settings.

9.1 Ambiguous Shortest Paths

In our exposition we always assume uniqueness of shortest paths. In this section we will discuss necessary modifications in case shortest paths are ambiguous.

First of all, we can enforce uniqueness of shortest paths by symbolic perturbation. To that end we define the cost of a path $\pi = v_0 v_1 \dots v_k$ not only as the sum c_π of its edge costs, but by the vector $(c_\pi, v_0, v_1, \dots, v_k)$. Two such cost vectors are compared lexicographically, that is, if two $s-t$-paths $\pi_1 = sv_1 \dots t$ and $\pi_2 = sw_1 \dots t$ have the same aggregated edge costs, and i is minimal with $v_i \neq w_i$, then π_1 is considered shorter if $v_i < w_i$, otherwise π_2 is considered shorter. This symbolic perturbation can easily be incorporated into e.g. Dijkstra's algorithm. During the computation of shortest paths from a node s we consider as (possibly tentative) distance label of a node v from s not only the aggregated edge costs $d_s(v)$ along the respective path but the tuple $(d_s(v), pred_s(v))$, where $pred_s(v)$ denotes the predecessor on the current path from s to v. The ordering in case of identical $d_s(v)$ values is determined by the node ID of the predecessor. Edge relaxations as well as the organization of the priority queue is made according to these augmented distance labels. It is easy to see that this yields the canonical and unique shortest paths as described above.

With edge lengths typically measured at a precision of around one meter it rarely happens that two paths exhibit exactly the same length. Under some circumstances, though, it might be desirable to actually maintain multiple shortest paths between nodes (hitting each of them / allowing to travel along each of them without running out of energy). Fortunately, we can adapt our algorithms to cater for *all* shortest paths. There is a minimal change when backtracking after a Dijkstra exploration as well as in the PNM approach and a slight change in the CH construction. For the former, when we retrieve paths/sets, instead of following a predecessor reference of a node v (which was set during edge relaxation), we inspect all adjacent nodes and check whether their distance labels with the respective edge cost sums up to the distance label at v. This yields all neighbors of v that lie on a shortest path from s to v. Recursing on these neighbors we obtain all shortest paths. In the CH construction, the crucial operation is the contraction of a node v. In the original version, for every pair of neighbors u, w of v with $(u, v), (v, w) \in E$, a shortcut (u, w) is created (with cost of the path uvw) if uvw is the *only* shortest path from u to w. To maintain all shortest paths, we add the shortcut if uvw is a shortest path from u to w (but with possible existence of other shortest paths). In this way every shortest path has a CH representation; this comes at the cost of slightly more shortcuts added.

Our lower bound construction in Section 8.2 can be modified to also yield a lower bound for the case with ambiguous shortest paths as follows: An s-t-pair can contribute to the lower bound if all shortest paths between s and t require at least one recharging event. To compute a valid lower bound we retrieve a maximal set of such vertex pairs, where for any two vertex pairs in this set the respective shortest path node sets are not allowed to overlap. This generalizes the idea of node-disjoint shortest paths in case of ambiguous shortest paths.

Our experiments showed, though, that considering all shortest paths does not yield noticeably different results – mainly due to the rarity of reasonably long ambiguous shortest paths. When we disregard ambiguities in our implementation, only for extremely small battery capacities (corresponding to a cruising range of less than 2km), we found ambiguous shortest paths that were not covered by our BLS placement. For all larger battery capacities our BLS placement in fact covered all (of the few) ambiguous shortest paths.

9.2 Restricted Loading Station Placement

We assume in our ESC problem definition that loading stations can be placed on every node in the network. In practice, though, the set of possible locations might be restricted due to technical, financial or legal reasons. If there is a set $V' \subset V$ of candidate nodes for loading stations, we can incorporate this in our algorithms as follows: During construction of the set system, we check for every shortest path whether at least one of its nodes is in V'. Otherwise, we ignore the path completely (as it can never be hit anyway). For the CH-based extraction methods, we can set a flag for every edge/shortcut indicating whether the spanned path contains a node from V' or not. This allows to perform the check for a CH-path without having to unpack it. In the actual Hitting Set computation we simply skip over nodes that are not in V' to compute a feasible solution.

Note, that depending on the choice of V' the final Hitting Set might not allow to drive on all shortest paths without running out of energy, though. To incorporate that some locations are more suitable to become loading stations than others without losing global reachability as demanded by our ESC problem formulation, we introduce a prize function $p : V \to \mathbb{R}^+$. The higher the prize the more complicated or expensive it is to place a charging station at this node. Then we can exploit the *weighted Hitting Set* problem as basis for our computations. Here the goal is to find a set L of elements in the universe which hit all sets in the set system while minimizing the accumulated prize $\sum_{l \in L} p(l)$. Our set system extraction methods remain unaffected by the prizes. Only in the greedy Hitting Set computation step, the selection of the next best hitter changes. Previously, we selected the node next that hits most so far unhit sets. Now, if $S(v)$ denotes the set of so far unhit sets that contain v, we select the node that minimizes the average prize per set $p(v)/|S(v)|$. The approximation guarantee of the greedy algorithm for the weighted Hitting Set problem is the same as for the unweighted Hitting Set problem (Chvatal, 1979). We expect the solution to consist mainly of cheap charging stations and possibly some expensive ones that are required to establish reachability between any two nodes.

9.3 Placement with Given Initial Loading Station Set

Another assumption made in the ESC problem definition is that we try to construct a network of loading stations from scratch, i.e. starting with no loading stations at all. While loading stations are still sparse in many areas, the ones that are already installed should not be ignored. Existing loading stations can easily be taken into account when solving the ESC problem: For a given initial set of loading stations L_0, we check during the extraction of the set system for each path if it is already hit by L_0. If this is the case, the path can be pruned. The remaining steps for the Hitting Set computation work just like before.

10. Experimental Evaluation

Our proposed techniques for computing ESC solutions were evaluated in a multi-threaded implementation written in C++ and executed on 2nd generation Intel Core desktop hardware, an i7-3930 (6 cores, 64GB of RAM) for complete set generations and an i7-2700 (4 cores, 32GB RAM) for the multi-stage construction with nested Hitting Sets. We use the following abbreviations to state results: K=10^3, M=10^6, s=seconds, m=minutes, h=hours, d=days, GB=10^9Bytes. We distinguish between CPU time (total CPU usage) and real time (wall clock time). Several road networks of Germany derived from OpenStreetMap data

region	abb.	$\|V\|$	$\|E\|$
Pforzheim	(PF)	0.2M	0.4M
Tübingen	(TU)	0.5M	1.0M
Baden-Württemberg South	(BW)	2.2M	4.6M
Southern Germany	(SG)	4.2M	8.6M
Germany	(GE)	17.7M	36.0M

Table 1: Test graph characteristics.

(OSM, 2015) were used for evaluation, see Table 1 for an overview. As edge cost function c we used travel time along an edge, so the paths to hit are indeed quickest paths (the term *shortest* paths is conventionally used for subsuming all kinds of metrics). Energy consumption of an EV was modeled as explained in the introduction using distance data from OSM and elevations provided by the Shuttle Radar Topography Mission (NASA, 2015). B corresponds to a battery capacity which translates to a certain terrain dependent cruising range. We use a capacity B for PF and TU that allows to drive 40 kilometers on average, and about 125 kilometers for the larger graphs. Our α, which models how much going uphill increases the energy consumption, equals 4.

10.1 Dealing with Complete Set Systems

Construction and Representation. Let us first examine the time and space complexity of extracting the complete set of minimal B-violating paths. We constructed set systems using the naive strategy (*NAIVE* – representing each path as the complete sequence of its vertices), the PHAST-based exploration (*PHAST* – with paths in CH representation), and peak node mapping (*PNM* – representing each path as *source,peak,target* triple). The respective results can be found in Table 2. Unfortunately, only the two smallest instances were feasible to process using all strategies; already for the BW graph, the time and space consumption of NAIVE exploded (extrapolated more than 500GB and more than 23 CPU days). In comparison, PHAST is about a factor of 3 faster than NAIVE, and the space consumption of CH-paths is an improvement by at least an order of magnitude. PNM can construct the BW instance in 4.4 CPU hours, compared to the week needed by PHAST, and the space consumption using triples decreases by another factor of 2 (note that for longer paths the advantage of PNM vs. CH representation increases). But for SG and GE, also PHAST and PNM took too much time and space (e.g. extrapolated 557GB/112days for PHAST). So for larger networks, constructing complete set systems seems to be infeasible.

Comparison of Path Representations. As explained in Section 5.5, the path extraction scheme does not tie us to a path representation. Instead, we can transform the extracted

Graph	# paths	space consumption		
		NAIVE	PHAST	PNM
		vertex sets	CH-paths	triples
PF	38M	5.2GB	0.2GB	0.1GB
TU	168M	24.0GB	2.1GB	0.9GB
BW	2715M	[526.3GB]	34.6GB	14.3GB

Graph	computation time					
	NAIVE		PHAST		PNM	
	CPU	real	CPU	real	CPU	real
PF	1.5h	0.3h	26.7m	5.0m	1.8m	25.1m
TU	24.6h	4.1h	7.5h	1.4h	27.3m	5.4m
BW	[23.1d]	[3.9d]	7.5d	33.1h	4.4h	47.0m

Table 2: Comparison of path extraction/representation schemes. B corresponds to about 40km (PF and TU) or 125km (BW) cruising range on flat terrain. Timings include the CH-construction for PHAST/PNM. Values in brackets are extrapolated.

paths to any of the introduced representations before storing them. Each representation provides some trade-off between space consumption and access times for single paths. Figure 10 illustrates these values for a small and a large benchmark instance (TU and GER).

Note, that the access times for paths represented as triples are amortized. If we really want to extract a single path only, the costs are comparable to the ones for the (s, t) representation. But in the greedy algorithm, we require access to huge sets of paths in every round, so the (s, p, t) representation pays off. For the CH representation, the access times are reported in the figure purely for completeness. They are not significant for the greedy Hitting Set computation, though, as our specialized greedy algorithm on CH paths does not require to unpack those paths. In fact, the access times relevant for the greedy algorithm are even below the ones for the vertex set representation, as the CH representation contains far less elements we have to sweep over. Hence, we regard CH-paths as the best path representation as soon as the set system fits in memory in this representation.

The transformation times between two path representations can be estimated from the results reported in Figure 10 as well. Every transformation that runs in constant or linear time according to our analysis has cost less or comparable to accessing paths in vertex set representation. The transformation time for CH-paths to vertex sets or vice versa corresponds to the access time for paths in CH representation. And to transform triples into CH-paths, we need the time to access a path in triple form minus the time to unpack a CH-path.

Hitting Set Computation. We evaluated both the standard greedy algorithm as well as the multiple hitters (MH) variation on the set systems for PF, TU, and BW with varying choices for B. Figure 11 shows their performance in terms of quality (standard greedy vs. MH) as well as running time (how much faster MH is compared to standard greedy). The ratios are averaged over all test graphs; the bound B is chosen between almost zero and 60 percent of

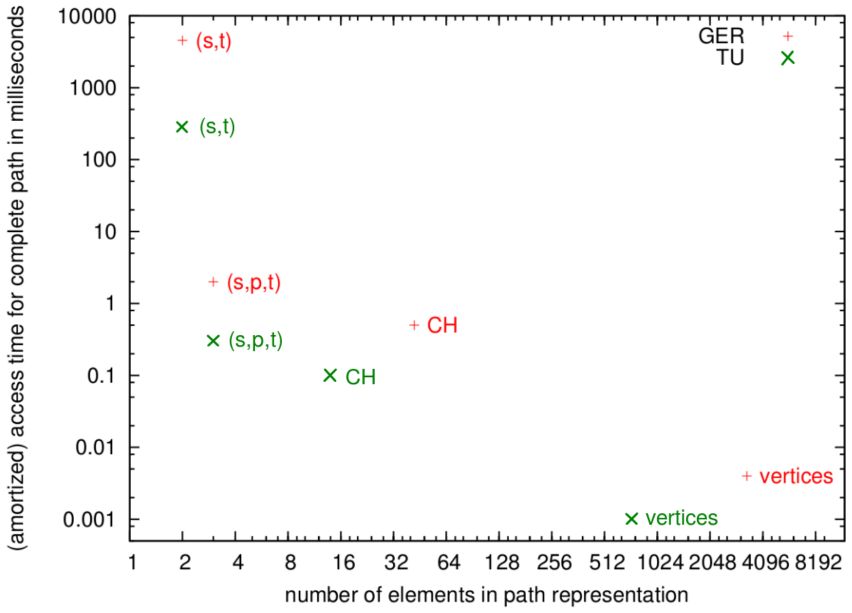

Figure 10: Comparison of several path representation schemes in terms of space consumption and access time. Both axes are in log scale.

the maximum energy consumption of some shortest path in the respective network (in fact for very long paths, the set systems got so simple that greedy even constructed the optimum, hence the approximation ratio of 1 in Figure 11). In all cases, greedy produces results much closer to the optimum as the theoretical $2 \ln n$ guarantee, the maximum deviation from the lower bound was indeed less than 4.5. Employing the MH strategy increases the HS size slightly, but yields significantly decreased running times especially for smaller bounds B (where more hitters have to be chosen). Still, compared to the construction time of the set systems, the Hitting Set computation times were negligible, so we do not state them explicitly here. This will change when we employ the multi-stage construction, though.

10.2 Multi-stage Construction

As the construction of the complete set system has proven to be infeasible for larger road networks, we will make use of the idea of a multi-stage construction.

k-Hop Cover+PNM. Let us first examine how a compact set system can be constructed using the PNM approach after an initial k-Hop Path Cover. For the BW network we computed a $k = 32$-Hop Cover C (146,494 nodes) which corresponds to an ESC solution with $B' = 8832$ (and cruising range of about $9km$ in flat terrain). Then PNM is used to create a final compact set system by only considering $(B - B')$-violating paths that start at nodes in C. Not surprisingly, the number of paths to be hit reduces drastically from 2715M in Table 2 to 24M in Table 3. The running times are still quite high, though, as this approach does not save the exploration from each peak (therefore more stages will not

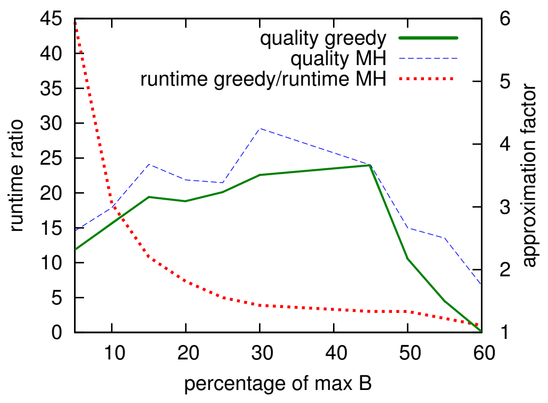

Figure 11: Performance of the greedy algorithm and the multiple hitters variant (MH) averaged over PF, TU and BW.

Graph	$\|C\|$	B'	CPU	real	# paths
BW	146,494	8832	2.2h	35.5m	24M
SG	180,455	10048	6.0h	2.8h	75M
GE	769,760	15808	64.8h	27.6h	1085M

Table 3: Instance creation ($B = 40K$) via PNM with initial k-hop solution C for $k = 32$.

help much here). Since further improvements in terms of running time using PNM in a multi-stage approach cannot be expected, let us now concentrate on the naive approach with the extracted paths being converted into their CH-representation.

Multi-Stage Hitting Sets. We employ the following strategy: first, construct a k-Hop Cover C with e.g. $k = 32$ which yields an initial Hitting Set L_r for some bound B_r. Then we construct a reduced set system consisting of all $(B_{r-1} - B_r)$-violating paths starting at nodes from L_r only and compute a Hitting Set L_{r-1} for that set system. Here, we use naive

Graph	B_r	B_i's of nested HS	$\|L\|$	LB	APX	CPU	real
TU	–	1k,5k,40k	120	33	3.64	9m	3m
SG	–	2k,10k,125k	106	33	3.21	404m	106m
TU	1.8k	4.8k,40k	116	33	3.52	8m	2m
SG	4.2k	12.2k,125k	110	33	3.33	242m	63m
GE	15.8k	17.8k,33.8k,125k	868	190	4.57	908m	265m
GE	6.2k	8.2k,24.2k,125k	728	190	3.83	1156m	322m
GE	15.8k	17.8k,25.8k,49.8k,125k	1212	190	6.38	645m	209m

Table 4: Multi-stage Hitting Set computation (LB = lower bound, APX = approximation factor). The last two experiments can be seen in detail in Table 5.

i	B_i	t_{SS}	#paths	t_{HS}	$\|L_i\|$	CPU
4	6.2k	–	–	14s	1388k	14s
3	8.2k	363m	34.6M	25m	223k	388m
2	24.2k	249m	36.8M	21m	16k	270m
1	125k	467m	13.5M	31m	728	498m
$\sum$		1079m	–	77m	–	1156m
i	B_i	t_{SS}	#paths	t_{HS}	$\|L_i\|$	CPU
5	15.8k	–	–	36s	770k	36s
4	17.8k	203m	19.7M	25m	208k	228m
3	25.8k	101m	17.1M	21m	38.6k	122m
2	49.8k	81m	9.2M	17m	8445	98m
1	125k	146m	5.6M	50m	1212	196m
$\sum$		531m	–	113m	–	645m

Table 5: Statistics for a 4-stage run starting with a $k = 16$-Hop Cover (above), and a 5-stage construction initialized with a $k = 32$-Hop Cover (below) on GE. The column *#paths* gives the number of sets to hit in the respective stage. t_{SS} and t_{HS} denote CPU time for the set system construction and the Hitting Set computation respectively.

extraction but transform vertex sets to CH-paths for more efficient storage. We proceed iteratively until reaching $B_1 = B$ and the final Hitting Set $L_1 = L$. It is intuitive to demand that the gap between B_2 and B_1 should be large to make sure that the last Hitting Set instance still faithfully characterizes the original Hitting Set instance. Table 4 shows the results for various choices of multi-stage parameters. In Table 5 we give a more detailed account on the intermediate calculations for the large GE graph. The experiments confirm that the larger the gap between B_2 and B_1 is, the better the quality of the final Hitting Set. This comes at the cost of an expensive last stage, though. In contrast to the other experiments, the first two calculations in Table 4 have been conducted without an initial k-Hop Cover. The results obtained on TU and SG suggest that an initial k-Hop Cover accelerates the calculation while maintaining a similar Hitting Set size. Furthermore, all the APX values remain low even though the lower bounds were obtained in a naive way. Note that for example for the graph of Germany, the a priori approximation guarantee of the plain greedy algorithm (which is not feasible due to excessive running time and space consumption) is $2 \ln n \approx 19.5$. This proves the excellent quality of the Hitting Sets for the particular instances. Table 5 shows that introducing multiple stages keeps the intermediate set systems rather compact, so efficient computation is actually possible.

11. Conclusions and Future Work

We showed how to model and solve a natural and important facility location problem in the E-mobility context, taking a radically different approach than previous ones avoiding detours to loading stations for EVs.

While a naive strategy only allows for the solution of small instances of few hundred thousand nodes, our compact representation schemes for the underlying set systems and heuristic modifications of the standard greedy approach make the computation of a solution even for country-sized networks like that of Germany possible. Instance-based lower bounds certify that the solution quality is close to optimal (within a factor of 4) and far from the pessimistic theoretically achievable approximation bound. In fact it is remarkable that after all, it was possible to compute a 4-approximate solution to a seemingly intractable Hitting Set problem within a few hours on a quadcore desktop PC. Our computation determined around 800 locations where placing BLSs would establish complete coverage for Germany.

Our framework does not require the metric that decides which shortest paths have to be hit to be identical with the metric that determines which paths are shortest. In fact this was factored out using our η function, which – depending on the application scenario – can also be used to implement other hitting criteria (e.g. hop distances or risk values).

In future work we intend to examine how the 'exact hitting' requirement can be relaxed. Naturally, it is not necessary that there is always a BLS right on the respective shortest path, but a nearby one suffices. This could be modeled by enlarging the vertex sets of the respective shortest paths by surrounding vertices. Hitting Set sizes for this variant are expected to be considerably smaller than for hitting all shortest paths directly. Another direction of research is to take into account capacity constraints of the BLSs (Lam et al., 2013); in particular in urban areas it is certainly necessary to provide recharging stations for a very large number of vehicles.

References

Abraham, I., Delling, D., Goldberg, A. V., & Werneck, R. F. (2012). Hierarchical hub labelings for shortest paths. In *European Symposium on Algorithms (ESA)*, pp. 24–35. Springer.

Artmeier, A., Haselmayr, J., Leucker, M., & Sachenbacher, M. (2010). The shortest path problem revisited: Optimal routing for electric vehicles. In *German Conference on Artificial Intelligence (KI)*, pp. 309–316.

Arz, J., Luxen, D., & Sanders, P. (2013). Transit Node routing reconsidered. In *International Symposium on Experimental algorithms (SEA)*, pp. 55–66. Springer.

Bast, H., Funke, S., & Matijevic, D. (2009). Ultrafast shortest-path queries via Transit Nodes. In *The shortest path problem : 9th DIMACS implemenation challenge*, Vol. 74 of *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*, pp. 175–192, Providence, RI. AMS.

Berg, M. d., Cheong, O., Kreveld, M. v., & Overmars, M. (2008). *Computational Geometry: Algorithms and Applications* (3rd ed. edition). Springer-Verlag TELOS, Santa Clara, CA, USA.

Chvatal, V. (1979). A greedy heuristic for the set-covering problem. *Math. of Oper. Res.*, *4*(3), 233–235.

Delling, D., Goldberg, A. V., Nowatzyk, A., & Werneck, R. F. F. (2011). PHAST: Hardware-accelerated shortest path trees. In *International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 921–931.

Dinur, I., & Safra, S. (2004). On the hardness of approximating minimum Vertex Cover. *Annals of Mathematics*, *162*, 2005.

Eisner, J., & Funke, S. (2012). Transit Nodes - lower bounds and refined construction. In *Algorithm Engineering and Experiments (ALENEX)*.

Funke, S., Nusser, A., & Storandt, S. (2014a). On k-Path Covers and their applications. In *International Conference on Very Large Databases (VLDB)*.

Funke, S., Nusser, A., & Storandt, S. (2014b). Placement of loading stations for electric vehicles: No detours necessary!. In *AAAI Conference on Artificial Intelligence*.

Geisberger, R., Sanders, P., Schultes, D., & Delling, D. (2008). Contraction Hierarchies: faster and simpler hierarchical routing in road networks. In *International Workshop on Experimental Algorithms (WEA)*, pp. 319–333. Springer.

Lam, A., Leung, Y.-W., & Chu, X. (2013). Electric vehicle charging station placement. In *International Conference on Smart Grid Communications (SmartGridComm)*, pp. 510–515.

NASA (2015). Shuttle Radar Topography Mission. Online. http://www2.jpl.nasa.gov/srtm.

OSM (2015). The OpenStreetMap Project. Online. http://www.openstreetmap.org.

Storandt, S., & Funke, S. (2013). Enabling E-Mobility: Facility location for battery loading stations. In *Conference on Artificial Intelligence (AAAI)*.