Institute of Formal Methods in Computer Science
University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelor's Thesis Nr. 38

EV Hitting Sets in Road Networks

André Nusser

Course of Study: Computer Science

Examiner: Prof. Dr. Stefan Funke

Supervisor: Prof. Dr. Stefan Funke

Commenced: November 21, 2012

Completed: May 23, 2013

CR-Classification: G.2.2

Abstract

As electric vehicles (EVs) become more and more popular, there also has to exist an appropriate infrastructure of battery loading stations to allow for a widespread usage. Especially long distance routes are still not covered, due to the short cruising range of EVs. In this thesis we develop an algorithm for placing such stations so that every shortest path can be driven without running out of energy, assuming an adjustable initial and maximum battery charge. Considering an initial roll-out of battery loading stations, we aim at placing as few as possible, while still meeting the above constraint. Therefore, we rely on a theoretical hitting set formulation of the problem to be able to precisely analyze and evaluate it, followed by a – at first – naive algorithm which is then improved in the course of the thesis. A dual problem is introduced to allow a computation of instance-based bounds. Finally we evaluate our implementation practically in regard to memory usage, runtime and quality of the results and furthermore theoretically prove general upper bounds. The final algorithm is capable of computing a battery loading station positioning on the graph of Germany in less than one day on our testing machine, with evidentially good quality.

Contents

1 Introduction 9

2 Basics 15
2.1 Electric vehicle routing . . . 15
2.2 Set systems, hitting sets and set packing . . . 16
2.2.1 Hitting sets . . . 16
2.2.2 Set packing . . . 17
2.2.3 Duality . . . 17
2.3 Contraction hierarchies . . . 18

3 Positioning of battery loading stations 21
3.1 Shortest path set system . . . 22
3.2 Positioning as a hitting set problem . . . 23
3.3 Basic algorithm . . . 24

4 Improvements 27
4.1 Space reduction . . . 28
4.1.1 Improved set system structure . . . 28
4.1.2 Contraction hierarchies . . . 30
4.2 Runtime reduction . . . 32
4.2.1 Incremental hitting set construction . . . 32
4.2.2 Multiple hitters . . . 34
4.2.3 Parallelization . . . 35

5 Analysis 37
5.1 Memory usage . . . 38
5.2 Calculation time . . . 38
5.3 Approximation bounds . . . 41

5.3.1 Theoretical bounds 41
5.3.2 Instance-based bounds 44
5.4 Quality 44

6 Conclusion and future work 47

A Appendix 51

Bibliography 53

List of Figures

1.1 Positioning of BLSs on a small graph with reduced cruising range. 10
1.2 Example of EV routing. 12

2.1 Examples of a node contraction and a CH path. 19

3.1 The difference in length two minimally ev-infeasible paths might exhibit. 22
3.2 Overlay of the paths of a SPSS with the placed BLSs. 23

4.1 Visualization of the data structures for the set system, with the overhead data marked red. 29
4.2 Example of the partial independence of greedy hitting. 35

5.1 Visualization of the performance of the incremental approach 39
5.2 There is no upper bound of the incremental hitting set, using $|\mathcal{H}(\mathcal{S}_M)|$. 42
5.3 An example of a computed BLS positioning. 46

List of Tables

5.1 Overview of the test results of the final implementation. 38
5.2 Comparison of the test results of different approaches. 38
5.3 Results of multiple hitters tests. 41

List of Algorithms

3.1 The basic BLS positioning algorithm 25

4.1 The CH BLS positioning algorithm 31
4.2 Multiple hitters adaption 35

5.1 Practical packing algorithm 44

CHAPTER 1

Introduction

Electric vehicles (EVs) are a serious alternative to traditional, gasoline-driven vehicles, offering great advantages in many ways. Probably the most known benefit is the independence of fossil fuels, shifting the production of energy to other, more future-oriented fields. By changing the way of powering a vehicle, the degree of efficiency is increased notably. Also, EVs allow for an energy recuperation when driving on descending roads, thus reducing the overall energy usage. Furthermore, the by-products of an EV can be recycled more easily, as they are not released to the environment in an uncontrolled process. Not only are there many environmental benefits, but the widespread usage of EVs would also greatly reduce the traffic noise, thus improving the quality of life in many areas and supersede the construction of traffic noise protection in many instances. In spite of these major advantages, EVs are still not very present on the roads due to long recharging times and generally higher car prices. Also the reach is below the average of a 'normal' car and the sparse grid of battery loading stations (BLSs) amplifies this disadvantage. Especially concerning long distance routes, progress in the positioning of BLSs is very urgent, as the efforts have mainly been focused on short distance routes. Therefore, we want to develop a practical algorithm, that computes a minimal positioning of BLSs, that allows a common trafficability of the road network which is given as a graph. To achieve this goal, we have to use a mathematical model for EVs and combine it with a model of a desired BLS positioning, while ensuring that the resulting computation is practical applicable. Figure 1.1 shows an example of a result of the final implementation.

To abstractly specify the battery consumption in a graph $G = (V, E)$, we need a function $\eta : E \to \mathbb{R}$. Either η is given as a black box or we might construct it ourselves by taking energy characteristics of a vehicle and the elevation profile of the area into account. In practice we derive it from the height of the source and target nodes and the euclidean length of the edge.

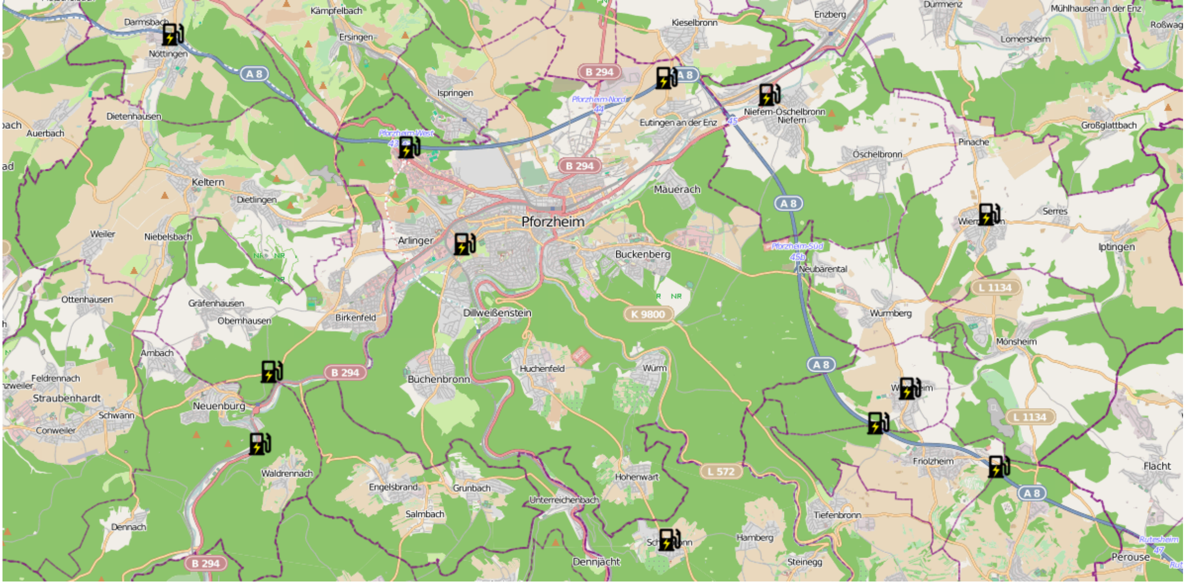


Figure 1.1: Positioning of BLSs on a small graph with reduced cruising range.

We first introduce symbols for these two properties in order to define the battery consumption function thereafter.

Definition (edge-length $\ell(e)$)**.** *The length of the edge $e \in E$ is denoted by $\ell(e)$.*

Definition (node-height $h(v)$)**.** *The height of the node $v \in V$ is denoted by $h(v)$.*

Definition ($\eta : E \to \mathbb{R}$)**.**

$$\eta(e = (s,t)) = \begin{cases} \alpha \cdot \ell(e) + h(t) - h(s), & h(s) < h(t) \\ \alpha \cdot \ell(e) + \beta \cdot (h(t) - h(s)), & \textit{else} \end{cases}$$

for some $\alpha, \beta \geq 0$.

The parameter α controls the ratio between the battery consumption by length and by height difference. With β we regulate the amount of energy we might regain when taking downwards edges.

To apply this model to paths, we additionally need to know the battery capacity M which defines the initial and also the maximum battery charge. Note that the battery naturally cannot be over or undercharged. Thus, when $b(x)$ denotes the current battery charge an EV has on node $x \in V$, then $b(x) \in [0, M]$ must hold. To facilitate the notation we define $b(x)$ to be $-\infty$ when the node x cannot be reached and thus $b(x) \in \mathbb{M}$ with $\mathbb{M} = [0, M] \cup \{-\infty\}$. Following from this we define the transition function $\tau : E \times \mathbb{M} \to \mathbb{M}$.

Definition ($\tau : E \times \mathbb{M} \to \mathbb{M}$)**.**

$$\tau(e,m) = \begin{cases} -\infty, & m - \eta(e) < 0 \\ M, & m - \eta(e) \geq M \\ m - \eta(e), & else \end{cases}$$

Now that the usage of an edge for EVs is defined, we extend this definition to whole paths. The *ev-feasibility* of a path, with respect to a battery capacity M, states, whether we can take the path without the battery running empty.

Definition (ev-feasible)**.** *A path* $p = e_1 \dots e_k, e_i \in E$ *is ev-feasible if:*

$$\tau(e_k, \tau(e_{k-1}, \dots \tau(e_1, M) \dots)) \geq 0$$

We later use various path representations and naturally extend the definition of *ev-feasible*, by implicitly applying the definition to the edge representation. Furthermore, an ev-*in*feasible path p is *minimal*, when every strict sub-path of p is ev-feasible.

One might assume that the differences between EV routing and distance based routing are marginal, but a closer look proves this assumption to be wrong. There are two major differences. The first one is, that in an EV routing environment, negative edges occur frequently, while they seldom exist in distance based routing and are forbidden if we use Dijkstra's algorithm [DIJ59] to find shortest paths. Although routing algorithms with negative edges are well researched, this property leads to the second difference, which we will examine in Figure 1.2. Resulting from the inability to over or undercharge the battery and the fact that once we run out of energy no further steps are possible, the ordering of the edges on a path matters greatly in regard to ev-feasibility.

Consider the three paths in Figure 1.2, where we start at one of the nodes u, v, w and drive counter clockwise until we return to the starting node; namely the paths $uvwu$, $vwuv$ and $wuvw$. Furthermore, we assume a battery capacity of $M = 10$. To test for ev-feasibility of the paths, we apply its definition:

$$\begin{aligned} uvwu: &\quad \tau(wu, \tau(vw, \tau(uv, 10))) &&= \tau(wu, \tau(vw, 10)) &&= \tau(wu, 2) &&= -\infty < 0 \\ vwuv: &\quad \tau(uv, \tau(wu, \tau(vw, 10))) &&= \tau(uv, \tau(wu, 2)) &&= \tau(uv, -\infty) &&= -\infty < 0 \\ wuvw: &\quad \tau(vw, \tau(uv, \tau(wu, 10))) &&= \tau(vw, \tau(uv, 4)) &&= \tau(vw, 9) &&= 1 \geq 0 \end{aligned}$$

Though we took the same edges for each path, the ev-feasibility differs. While the first two paths are infeasible, the third one can be driven without the battery running empty. Also, the first path is infeasible after three edges, while it takes the second path one edge less to lose its feasibility. These differences arise from the ordering of the edges. When we drive on the path $uvwu$, the edge that would allow for a recharge is taken too early and thus no recharging is possible and we run out of energy on the subsequent two edges. For the path $vwuv$ the opposite is the case, meaning that the edge with energy recuperation is at the end of the path, where we already ran out of energy. Naturally, the best time to take the negative edge uv is when we already used some energy, but still had something left to reach u, as with the path $wuvw$.

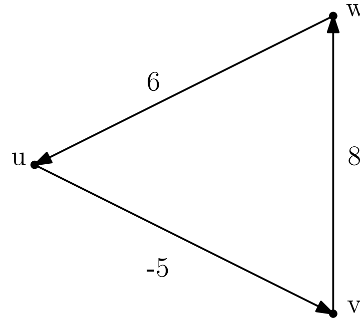


Figure 1.2: Example of EV routing.

Related Work

To our knowledge, the algorithmic routing problems of electric vehicles were first investigated in the year 2010 by [AHLS10], introducing energy-efficient routing which calculates the most energy-efficient path from one point to another. The runtime of the presented algorithm is in $O(n^3)$. This approach was then improved by [EFS11], obtaining a runtime of $O(n\log n + m)$ after an $O(nm)$ preprocessing step. The definitions of *ev-reachable* and *ev-connected* nodes, that we introduce in section 2.1, were made in a subsequent work [SF12]. In this publication battery switch stations, which play a big role in this thesis, were first introduced into this algorithmic context as well. Throughout the thesis we talk of battery loading stations (BLSs) instead of battery switch stations, meaning the same theoretical concept. Since we combine EV routing with shortest path computations, also Dijkstra's algorithm [DIJ59] is used.

Concerning the positioning of reloading or refueling stations, there are many approaches that are less algorithmic and are rather taken from a transportation science point of view. In [KLS+09] a model for the initial roll-out of hydrogen stations is described, using the example of Florida. Contrary to our approach, they assume a fixed number of stations that should be placed and try to find the most relevant ones. Furthermore, they use traffic analysis data for their model, while we rely only on the data of the road network. In [MRMZ11] the hydrogen station positioning model is extended towards a battery switch station positioning, using several areas of Australia as their testing instances. Again, a more complex model is developed to prioritize the positioning possibilities. But since we are mainly interested in an algorithmic approach that uses a simple, yet effective, model to obtain the positioning, these works are not used in the subsequent chapters.

Our contribution

We developed an algorithm to place BLSs, such that every shortest path can be driven with an arbitrary, but fixed, initial and maximum battery charge. The implementation can calculate a positioning of about $1,500$ stations on the graph of Germany in less than a day, using less than 50GB of memory, therefore being practical. To achieve this goal, we developed and applied

several speedup and memory reduction methods. Furthermore, we assured that the quality of the result is satisfying in different ways. Full parameterization is made possible by the implementation, making it easy to apply different models and thus analyzing the problem from different angles.

Structure

The work is structured in the following way:

In chapter 2, we give a short introduction to all the topics we later need in the main part of the thesis. The topics are definitions for EV routing, set systems, hitting sets and set packing. Furthermore, we give an introduction to contraction hierarchies, a technique recently developed in order to speedup shortest path computations. Then in chapter 3, we introduce the main topic of the thesis – the positioning of the BLSs. At this point we give an overview of the algorithm, that we use to compute a set of nodes that represent the stations. In chapter 4, we present ideas on how to reduce the space consumption and the runtime of the algorithm, in order to apply it to bigger graphs, and show their practical implementations. After that, in chapter 5, we analyze the algorithm to see what it is capable of. We analyze practical as well as theoretical aspects, and compare the practical results before and after the improvements, introduced in chapter 4, were applied. Finally, in chapter 6, we consider the results of the previous chapters and draw a conclusion, where we also have a look at possible future works.

CHAPTER 2

Basics

2.1 Electric vehicle routing

Given a graph $G = (V, E)$, two interesting aspects of EV routing are *ev-reachable* and *strongly ev-connected* node sets introduced in [SF12].

Definition (ev-reachable node set $R(s)$)**.**

$$R(s \in V) = \{t \in V | \exists p = s \dots t : p \textit{ is an ev-feasible path}\}$$

The ev-reachable node set $R(s)$, for $s \in V$, is defined as the set of all nodes that can be reached without completely draining the battery, while starting with a fully charged battery at node s.

Definition (strongly ev-connected node set $C(s)$)**.**

$$C(s \in V) = \{t \in V | \exists p = s \dots t \dots s : p \textit{ is an ev-feasible path}\}$$

The strongly ev-connected node set $C(s)$, for $s \in V$, is a subset of the ev-reachable node set $R(s)$. It contains only those nodes of R(s) from which we can return to s, using the remaining battery charge only.

These two definitions describe fundamental properties of EV routing. When $\exists v \in V : R(v) \neq V$, meaning we can not reach each other node from every arbitrary node, then EV routing is unpractical on this graph. Practicability can be restored by placing BLSs. Strong ev-connectivity, for all nodes $v \in V$, is an even stronger constraint and is not guaranteed to be met throughout the thesis.

2.2 Set systems, hitting sets and set packing

Both, the hitting set and the set packing definition, rely on set systems, thus they have to be defined first.

Definition (set system)**.** *A set system $\mathcal{S}$ over a universe $\mathcal{U}$ is a set of subsets for which $\forall s \in \mathcal{S} : s \subseteq \mathcal{U}$.*

In the subsequent chapters we interpret the $s \in \mathcal{S}$ as paths consisting of either nodes or edges.

2.2.1 Hitting sets

Definition (hitting set)**.** *A hitting set $\mathcal{H}(\mathcal{S}) \subseteq \mathcal{U}$, of a set system $\mathcal{S}$ over $\mathcal{U}$, is a set for which $\forall s \in \mathcal{S} : s \cap \mathcal{H}(\mathcal{S}) \neq \emptyset$.*

For every set system $\mathcal{S}$ there exists a trivial hitting set $\mathcal{H}(\mathcal{S}) = \bigcup_{s \in \mathcal{S}} s$. But the decision problem, asking if there exists a hitting set of size smaller than a given k, called the *hitting set problem*, is NP-hard [Kar72]. It follows that the *minimum hitting set problem*, which asks for the smallest possible k, might not be solvable in a practical setting due to its runtime. A problem that might be solved easier in practice is the *minimal hitting set problem*, where we only ask for a hitting set that minimizes the k locally, meaning that no $h \in \mathcal{H}(\mathcal{S})$ can be deleted without destroying the hitting set property.

The minimum hitting set problem can also be formulated as an integer linear program:

$$\begin{aligned} \text{minimize: } & \sum_{u \in \mathcal{U}} x_u \\ \text{subject to: } & \sum_{u \in s} x_u \geq 1, \forall s \in \mathcal{S} \\ & x_u \in \{0, 1\}, \forall u \in \mathcal{U} \end{aligned}$$

where $\mathcal{U}$ is the universe, $\mathcal{S}$ is the set system and x_u is an indicator variable that equals 1 if and only if u is in the hitting set [EF12]. This is useful to get the approximation quality of an instance through the dual linear program that is presented in 2.2.2.

In the algorithm that is developed in the subsequent chapters, we only approximate the hitting set with a greedy algorithm, without guaranteeing the property of a minimum or minimal hitting set. The greedy algorithm proceeds in rounds. In every round we choose the $u \in \mathcal{U}$ which maximizes

$$\varphi(u) = |\{s \in \mathcal{S} | u \in s\}|$$

put it into the hitting set and delete all the sets $s \in \mathcal{S}$ it is an element of. The algorithm terminates when all elements of $\mathcal{S}$ have been deleted. While in theory a greedy hitting set is a $\log n$-approximation [Chv79], we will show that the practical performance is much better in our setting. To facilitate the notation, we make the following two definitions:

Definition ($\mathcal{H}_g(\mathcal{S})$)**.** *$\mathcal{H}_g(\mathcal{S})$ denotes a greedy hitting set of the set system $\mathcal{S}$.*

Definition ($\mathcal{H}_{min}(\mathcal{S})$)**.** *$\mathcal{H}_{min}(\mathcal{S})$ denotes a minimum hitting set of the set system $\mathcal{S}$.*

2.2.2 Set packing

Definition (set packing)**.** *A set packing $\mathcal{P}(\mathcal{S}) \subseteq \mathcal{S}$ of a set system $\mathcal{S}$ is a set for which $\forall p, q \in \mathcal{P}(\mathcal{S}) : p \neq q \Rightarrow p \cap q = \emptyset$.*

There exists a trivial set packing $\mathcal{P}(\mathcal{S}) = \emptyset$. Because of that, similar to the hitting set, we are interested in testing if there exists a set packing with size greater or equal to a given k. This is called the *set packing problem*. It follows that there also exists a *maximum set packing*, which is a set packing $\mathcal{P}(\mathcal{S})$ that maximizes k globally. Both the set packing problem and the calculation of the maximum set packing are NP-hard.

The integer linear program formulation of the maximum set packing problem is:

$$\begin{aligned} \text{maximize: } & \sum_{s \in \mathcal{S}} y_s \\ \text{subject to: } & \sum_{u \in s} y_s \leq 1 \text{ }, \forall u \in \mathcal{U} \\ & y_s \in \{0, 1\}, \forall s \in \mathcal{S} \end{aligned}$$

where $\mathcal{U}$ is the universe, $\mathcal{S}$ is the set system and y_s is an indicator variable that equals 1 if and only if s is in the set packing [EF12].

2.2.3 Duality

The minimum hitting set problem and the maximum set packing problem are dual. This means that the solution of the dual (set packing) is a lower bound for the primal (hitting set). This gives us the possibility to derive instance-based lower bounds for a hitting set and thus calculate an approximation ratio – one of the main quality attributes of a hitting set.

We can recognize the duality between those two problems, when we transform a solution from the dual to the primal and vice versa. Given a set packing $\mathcal{P}(\mathcal{S})$ based on the set system $\mathcal{S}$, we know that each $p \in \mathcal{P}(\mathcal{S})$ has to be hit by at least one element of every corresponding hitting set $\mathcal{H}(\mathcal{S})$. Furthermore, we know that all the sets in $\mathcal{P}(\mathcal{S})$ are pairwise disjoint and therefore all have to be hit by different $h \in \mathcal{H}(\mathcal{S})$. It follows that $|\mathcal{P}(\mathcal{S})| \leq |\mathcal{H}(\mathcal{S})|$. Now presume that we have a hitting set $\mathcal{H}(\mathcal{S})$ and want to derive an upper bound of the corresponding set packing $\mathcal{P}(\mathcal{S})$. We know that every $s \in \mathcal{S}$ has to contain at least one node of $\mathcal{H}(\mathcal{S})$. Therefore, if we take out a set s to put it into our set packing, we can not put other sets with the hitters contained in s into $\mathcal{P}(\mathcal{S})$. We can do this step a maximum of $|\mathcal{H}(\mathcal{S})|$ times. It follows that $|\mathcal{H}(\mathcal{S})| \geq |\mathcal{P}(\mathcal{S})|$ and thus we are able to tell that the two problems are dual.

2.3 Contraction hierarchies

A *contraction hierarchy* (CH) is built from a graph – a preprocessing technique allowing very fast shortest path queries and even bringing more advantages, like the possibility to represent shortest paths with much less memory [GSSD08]. It was introduced in 2008 and has since then been regularly used to improve graph algorithms like the *single-source shortest path problem* [DGNW12] and the determination of *alternative routes* [ADGW10]. While the calculation of contraction hierarchies takes some time, it is usually outshone by the main computations that they are used for. Also, they need to be calculated only once for every graph and can then be stored, as the space of a contraction hierarchy typically does not exceed the memory usage of the source graph [GSSD08].

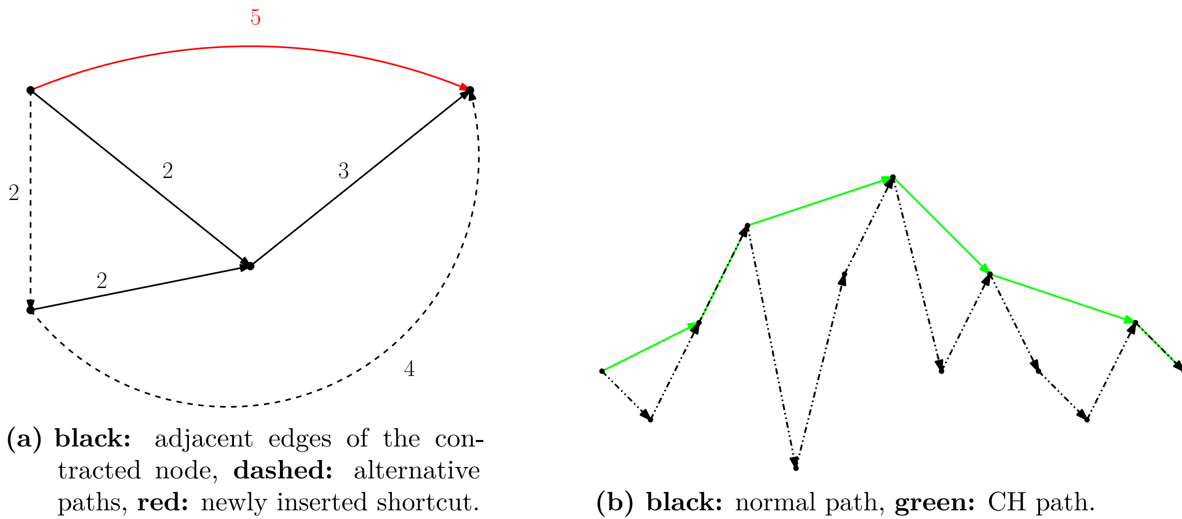
When constructing a contraction hierarchy, the main idea is to iteratively construct an overlay graph G' out of the previous graph, starting with the initial graph G. In every step nodes and edges are deleted from the previous one, while still preserving all the shortest path distances of the nodes which still exist in G'. This is done by adding *shortcuts* that represent the former shortest paths running over the node v we currently want to delete. The shortcuts that need to be added can easily be determined by performing a many-to-many Dijkstra query – from all the source nodes of the ingoing edges to the target nodes of the outgoing ones – and test which of the particular shortest paths run over v. For these, we have to insert shortcuts that preserve the shortest path. This process is called *node contraction*. See 2.1a for an example, where the center node is about to be contracted. As the shortest path between the upper two nodes runs over the node which is to be contracted, we have to insert a shortcut between them. The other possible shortcut is not inserted, since the shortest path from the lower left to the upper right node runs over the alternative path with length 4.

The point in time when a certain node is contracted is of importance for the shortest path queries and therefore we number the nodes ascending in the order they have been contracted. The related number of a node is then called it's *level*. It still remains to explain how we choose the node we want to contract next. The ideal order would be in ascending importance. This will be made clear in the explanation of the queries.

Contraction hierarchies have the very interesting property that every shortest path can be divided into two different parts such that the following holds. The first part only consists of *ascending edges*, meaning that the source node level is less than the target node level, while the second part solely contains *descending edges*, where the source node level exceeds the target node level[1] [GSSD08]. See Figure 2.1b for an example of a "normal" path and its corresponding CH path. Note that the "normal" path has twice as many edges as the CH path.

This leads to a very fast modified Dijkstra query, because we only need to search on ascending, outgoing edges from the source node and on descending, ingoing edges from the target node; in both cases in the manner of Dijkstra's algorithm. We call the nodes where these two searches

[1] This actually does not hold for ambiguous shortest paths. We ignore it in the following, as it is a rather technical issue.



(a) black: adjacent edges of the contracted node, **dashed:** alternative paths, **red:** newly inserted shortcut.

(b) black: normal path, **green:** CH path.

Figure 2.1: Examples of a node contraction and a CH path.

meet *candidates*, and stop searching when each of the Dijkstra query's distance exceeds the current shortest path, leading over a candidate.

In most cases we are not only interested in the distance of the shortest path, but also in the edges that are taken, for example when we want to compute the battery usage of a path. Note that the battery usage function η was only defined on normal edges and not shortcuts. Since CH paths might contain shortcuts, we have to somehow extract the 'normal edges' from the shortest path on the CH. This is possible by recursively unpacking the shortcuts and replacing each of them with the two corresponding edges they replaced during the construction of the CH. This hierarchical nature of the edges can be further exploited in our setting, as will be shown in section 4.1.2.

CHAPTER 3

Positioning of battery loading stations

The positioning of BLSs on a graph $G = (V, E)$ only makes sense when some constraints, that define the solution, are formulated. Generally we want to obtain a grid of BLSs that is as sparse as possible, while still being fully practical. Ev-reachability between all node pairs (s, t) is the minimum constraint that has to be met. Otherwise the network would not be suitable for EVs. Though, we still need to introduce further constraints. Imagine a route from s to t that would take us two hours driving with a "normal" vehicle. With an EV it could be the case that it takes four hours, or even longer, to reach t from s because a different, longer route has to be taken, due to the positioning of the BLSs. This is rather unpractical.

While the above approach would keep the number of BLSs very small, which is certainly a reasonable goal, we still need to improve the paths that can be driven. A more natural construction of drivable paths would be to place BLSs such that they allow driving each arbitrary shortest path $P \subseteq E$, with $\sum_{p \in P} \ell(p) \geq c$, where c is some constant. Although this approach is superior to the previous one there is still one problem we did not consider. While the fuel consumption of gasoline-driven vehicles can be seen as roughly constant, the battery consumption of EVs can differ vastly; it can even be negative in cases of descending edges. The effect of this property is a great difference in length two minimal infeasible paths might exhibit. In Figure 3.1 an example is shown.

Consequently, we have to place the BLSs depending on the battery usage induced by the paths. We can apply our model of an EV to this problem and set an arbitrary initial charge level M, which is also the maximum charge level of the battery. Now the constraint should not rely on the length of a path, but rather the battery consumption while driving on the path. So instead of choosing a constant c, we now want to place the BLSs such that every shortest path can be driven, without completely draining the battery, when starting with an initial charge level M. In the following we consider how we can solve this algorithmically.

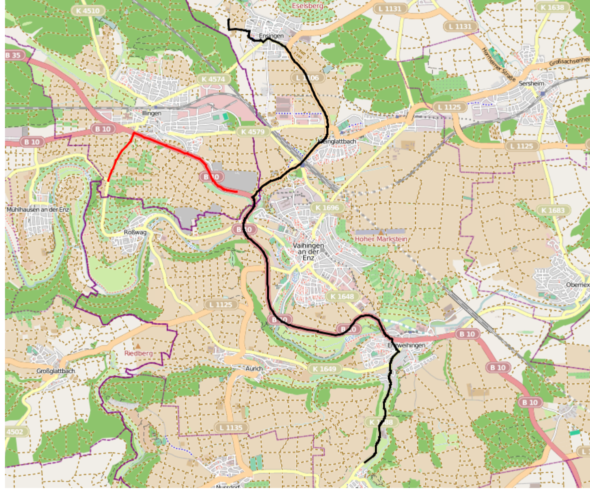


Figure 3.1: The difference in length two minimally ev-infeasible paths might exhibit.

3.1 Shortest path set system

First we want to extract all paths, that at least one BLS has to be placed on. These paths will be stored in a *shortest path set system* (SPSS). We already introduced and defined set systems in chapter 2.2. The definition of the SPSS adds further constraints.

Definition (shortest path set system (SPSS))**.** *Given a graph $G = (V, E)$, a SPSS $\mathcal{S}$ is a set system with elements $p_1, \cdots, p_k$ with $p_i \subseteq V, 1 \leq i \leq k$, that unambiguously define a path in G. Every path in the graph G that corresponds to a path in $\mathcal{S}$ has to be a shortest path.*

Note that if we really extract every shortest path that has to contain at least one BLS, we're also extracting a lot of redundant paths s' for which $s \subsetneq s'$ holds, for another path $s \in \mathcal{S}$. We call paths s for which

$$\forall s' \in \mathcal{S} : s \neq s' \Rightarrow s' \nsubseteq s$$

minimal violating paths as they are shortest paths, which are minimal in their violation of being drivable without recharging. The minimal violating paths must have the property, that we only need to place exactly one BLS on them, to make them non-violating. Otherwise, the problem is unsolvable, because there exists an edge that itself forms an ev-infeasible path.

To check for the minimal violation, we need a parameter M that defines the battery capacity of the EV. We assume that at the beginning of each path the battery is fully charged. Then we start a Dijkstra query at every node $v \in V$ and maintain the current battery charge (as described in chapter 1) for every element in the priority queue. Running the Dijkstra query until no node in the priority queue has a predecessor we could have reached without recharging, we can extract the shortest paths by backtracking the leaves of minimal infeasible paths. The leaves can be identified alongside the Dijkstra. Let $\mathcal{S}_M$ be the set system that arises from this procedure. Note that we have to remove the first node of the path, because by placing a BLS

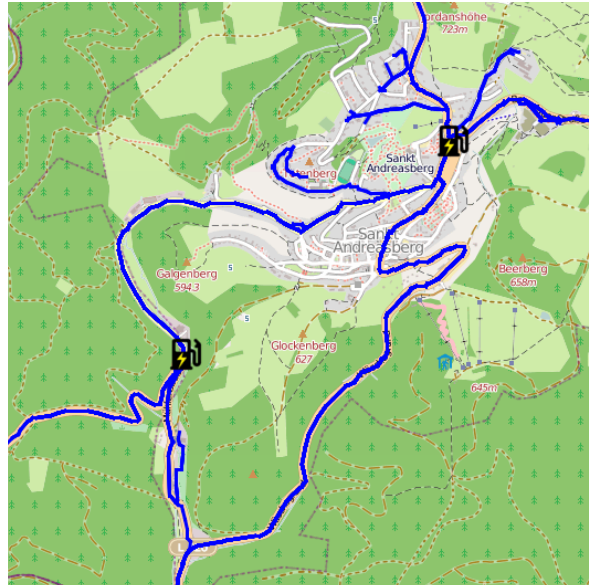


Figure 3.2: Overlay of the paths of a SPSS with the placed BLSs.

on this node, we do not make the corresponding path drivable. The same holds for the last node of the path. It boils down to the following definition:

Definition ($\mathcal{S}_M$)**.** *$\mathcal{S}_M$ is a SPSS of a graph $G = (V, E)$, containing all the shortest paths, without first and last node, that are minimally ev-infeasible with respect to a battery capacity M.*

Only extracting minimal violating paths is harder than one might think. For every path we would have to check if there exists another path, that is a prefix or a suffix of it. While a prefix test can be done during the construction of the SPSS, a practical suffix test would force us to reorder the elements, which is far from practical in the data structure we compulsorily use, as can be seen later. But as we avoid most non-minimal violating paths this does not have a great effect on the practicability of the algorithm.

In Figure 3.2 one can see an example of an overlay of the paths that form a SPSS. For visualization purposes we chose a small map with an adapted initial battery charge, where only long paths are taken into the set system because of boundary effects[1].

3.2 Positioning as a hitting set problem

Now that the SPSS has been calculated, we want to place the BLSs. Since we started with extracting every path we must place a BLS on and then limited ourselves to the minimal

[1]Many shortest paths are cut on the boundary and are thus not taken into the set system, as they can be driven without recharging.

infeasible paths, we end up with a hitting set problem where the hitters are nodes while the sets are the node paths. We want to hit every path $s \in \mathcal{S}$ with a node, meaning that we want to place a BLS on every path defined by a set in $\mathcal{S}$. We already introduced hitting sets in section 2.2.1. If we place the BLSs on the nodes of a hitting set $\mathcal{H}(\mathcal{S})$ of $\mathcal{S}$, we can drive every shortest path P of arbitrary length. This is the case, because we can partition P into the sections between the BLSs, the start and end node. These sections are assured to be ev-feasible, because either the shortest paths end without the battery running empty or a superset has to be contained in $\mathcal{S}$, and thus a BLS must have been placed before the battery runs empty.

As already mentioned in section 2.2.1, we compute a greedy hitting set, thus taking the node v into the hitting set that is an element of the highest number of sets in $\mathcal{S}$. We later introduce cases where the $s \in \mathcal{S}$ are not represented as node paths and where we first have to check somehow, which nodes correspond to the path of a single s. Note that this offers possibilities of speedups and the reduction of memory usage. We go into detail in chapter 4.

Because we always hit the most taken nodes when we follow the greedy strategy, in practice we will prefer crossroads and important highways. This lets us hope for useful practical results. Furthermore, the reduction of the problem towards a hitting set problem facilitates the analysis, as we can now easily argue on a mathematical, well studied foundation. In Figure 3.2 one can see an example of a hitting set with the corresponding SPSS.

3.3 Basic algorithm

Algorithm 3.1 shows the resulting basic algorithm for the calculation of a BLS positioning, where most of the detailed functionalities are omitted in favor of high level descriptions.

In lines 4 to 7 of algorithm 3.1 we compute the SPSS as described in section 3.1. The crucial part of this section is the calculation of the minimal infeasible paths in line 5. It will even dominate the runtime of the whole algorithm in the end, because the computation of the hitting set offers many speedup possibilities.

Then in lines 9 to 14 we calculate the greedy hitting set as described in section 3.2. In this section line 13 dominates the runtime. Though, by reducing the number of loop iterations, which is shown in 4.2.2, we can avoid a runtime that would be unfavorable. In practice we count the occurrences of the $v \in V$ in $\mathcal{S}_M$ only once and then update these numbers in the same pass as deleting the hit sets from $\mathcal{S}$.

Algorithm 3.1 The basic BLS positioning algorithm

Input: $G = (V, E)$, M **Output:** $\mathcal{H}_g$

```
1: S_M = ∅
2: H_g = ∅
3:
4: for all v ∈ V do                                        // Calculate the SPSS
5:     P = minimal infeasible paths, that start in v with initial battery charge M
6:     S_M = S_M ∪ P
7: end for
8:                                                          // Calculate the hitting set
9: while S_M ≠ ∅ do
10:    Count the occurrences of each v ∈ V in S_M
11:    v_max = node with the most occurrences in S_M
12:    H_g = H_g ∪ {v_max}
13:    S_M = S_M \ {s ∈ S_M | v_max ∈ s}
14: end while
15:
16: return H_g                                              // Return the computed hitting set
```

CHAPTER 4

Improvements

This chapter is divided into two parts. We first have a look at ways to reduce the memory usage of the implementation, which is possible by

- improving the data structure of the set system
- decreasing $|\mathcal{S}|$, the number of sets in the set system
- reducing the number of elements in the sets $s \in \mathcal{S}$

Since we maintain one huge set system, a goal is to avoid unnecessary overhead of the data structures, especially of the sets in the set system as they are numerous. Locality regarding the memory access leads to a reduced number of cache misses and therefore accelerates the data access. This can be achieved by saving and traversing the data in linear order, which itself is possible by storing the data in a single large array.

Another approach, the reduction of the number of sets in the set system, might be very hard, as we have to find sets that are prefixes or suffixes of other sets in the set system. For this we have to sort the set system, but since the overall computation is memory intensive, it might require too much additional memory. This is the case because it could be hard to sort the set system in place, depending on the utilized data structure.

That leads us to the reduction of the number of elements in sets of the SPSS, that can be achieved by changing the representation of the paths. When changing the representation, we also have to adapt the hitting set algorithm and therefore find a representation that compresses the paths, while the hitting set remains computable. Surprisingly, we can find a representation that not only reduces the memory usage drastically, but also accelerates the hitting set computation.

The second part of this chapter deals with the runtime reduction of the implementation. Possible approaches are

- splitting the set system calculation into subproblems
- further approximation of the hitting set
- parallelizing the implementation

The splitting of the set system calculation into subproblems is necessary, as we would otherwise have to compute shortest path trees of an unpractical size for many nodes. By introducing a round based set system calculation, that still assures the correctness of the algorithm, not only the calculation time is reduced, but also the memory usage.

Furthermore, the approximation of the hitting set yields a lot of possibilities to speedup the algorithm, while a good quality of the results has to be assured. One way is to modify the greedy hitting set algorithm, by not only taking the maximum node per round, but numerous others where we can assure that they hardly affect each other in the 'normal' greedy calculation.

Depending on the machine that we run the computation on, the parallelization might bring a huge speedup too. Since we use a 24 core machine for the evaluation, this improvement is mandatory. Of course it also depends on the possibility to parallelize the algorithm without many collisions.

4.1 Space reduction

4.1.1 Improved set system structure

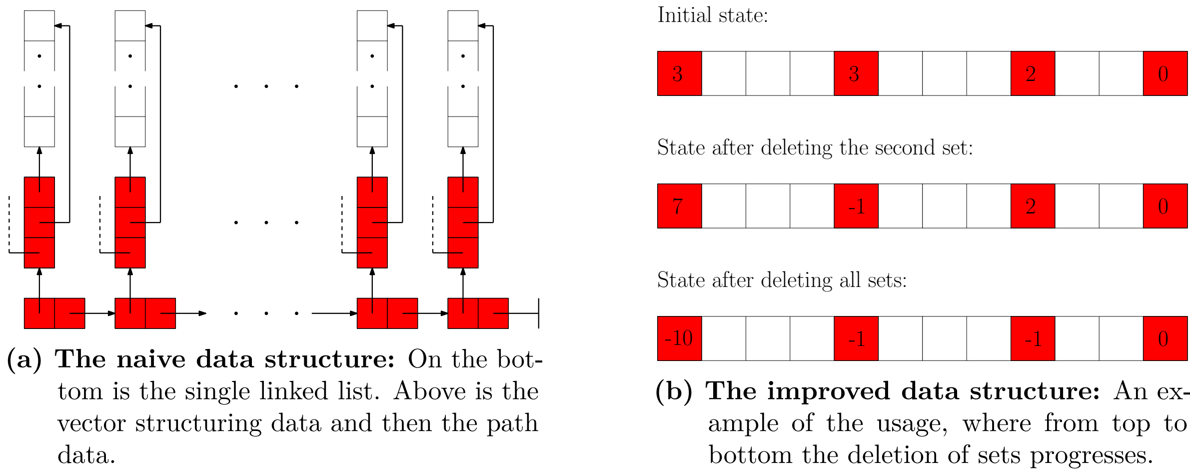
The first implementation of the set system was a single linked list of C++-vectors. The list as well as the vectors have a big memory overhead just for the data structure. The list keeps two pointers per element, while the vector keeps three[1]. So the overall overhead of the first data structure sums up to:

$5\,|\mathcal{S}| \cdot$ <size of a pointer>

A visualization of the data structure can be seen in 4.1a. On a 64-bit architecture, with the ids of the elements in the set system being only 32 bits long, the content of the set system only adds up to the structuring data, if the elements of the set system contain 10 elements in average. This overhead is not acceptable and thus we need to restructure the set system to construct one with less overhead, while preserving the flexibility we need.

For the new data structure the following points must be taken into account:

[1] The two pointers of the list are one to the next element and one to the data. The three pointers a vector is keeping are one pointer to the beginning of the allocated memory, one to the end of the allocated memory and one to the last element of the vector.



(a) **The naive data structure:** On the bottom is the single linked list. Above is the vector structuring data and then the path data.

(b) **The improved data structure:** An example of the usage, where from top to bottom the deletion of sets progresses.

Figure 4.1: Visualization of the data structures for the set system, with the overhead data marked red.

1. We do not want to use vectors because of their data overhead.
2. We do not want to use lists because of their data overhead.
3. It should be cache friendly, unlike lists.
4. We need a list-like data structure to remove sets at any position from the set system.
5. The structure needs to be designed for highly parallelized usage.

The logical consequence of not using vectors and lists is the usage of arrays. As we also know in which order we will traverse the sets, we can try to construct a single large array to keep all of them. Then points one to three of the requirement list would be met. Some length information of the path sets always has to be kept, but only if they have not been deleted yet. So the array could look like the first one in Figure 4.1b. For every path set we first store the size and then the set itself, and mark the end of the array by a ’0’, where another size element would have been placed otherwise.

No problem arises when extracting and storing the sets, but how can we imitate list-like behavior? If a set is deleted, we set its length value to -1 and thereby mark the boundaries between sets. From there on we want to skip it; the way it would be treated when it would have been deleted in a list. To achieve this, the length value of the previous element, that has not been deleted, is set to the number of array elements between itself and the length value of the next path that has not been deleted. This action can be seen in the second row of Figure 4.1b. The first length element forms a special case when deleted, because we want to skip the prefix of deleted sets of the big array in one step. Therefore, we do not set the length value to -1 when we delete it, but to the number of elements between itself and the first set that has not been deleted, times -1. This behavior can be seen in the last row of Figure 4.1b, where the length element of the first set points to the terminal ’0’. With this strategy, we can still

identify the boundaries of the sets and therefore the sets itself, while being able to skip sets that have been deleted.

Now there's only one property missing to make the new data structure practicable: the possibility of parallel usage. The naive data structure can be parallelized by introducing a semaphore for the list. If a thread wants to add a new element to the set system, it waits for the permission to enter the critical section and then inserts a new element into the list. The allocation of the vector and the insertion of the path does not have to be performed in the critical section, because we can do this while backtracking the path and then set the pointer of the list to the already created vector. Using this strategy, no bigger conflicts are expected. While this works great for the naive case, it can not be applied to the new data structure. The problem occurs when we want to put the already backtracked elements into the set system. As they have to be saved to a specific memory address due to the usage of a single large array, we must copy them using the same strategy as before. This is rather unpractical, because it increases the required time in the critical section and thus leads to a large amount of conflicts between the threads. The solution to this problem is straight forward. Every thread uses its own array to store its sets, thus no conflicts between the threads will occur when inserting a new path. Note that this also facilitates the hitting set algorithm that would have been impossible to arrange without conflicts.

Finally, the overhead of the new data structure sums up to

$$|\mathcal{S}| \cdot \text{<size of a length value>} + 1$$

because we only have one array element overhead per path plus the last zero-element. As we could choose *size of a length value* to be 32 bits in practice, the overhead is reduced by one order of magnitude in comparison to the naive data structure and therefore fully meets our expectations.

4.1.2 Contraction hierarchies

The second improvement that we want to apply to our algorithm are contraction hierarchies (CHs). They were introduced in section 2.3 and are a technique to speedup shortest path queries. Furthermore, CH paths are a compressed representation of shortest paths. We want to exploit the latter to reduce the overall size of the SPSS and maybe even accelerate the hitting set computation.

A CH path is much shorter because it has the maximum length $2 \cdot \max_{v \in V} \text{level}(v)$, whereas the theoretical maximum length of a normal Dijkstra path is $|E|$ and the practical is in $O(\sqrt{|E|})$. Contrary to the theoretical maximum level of a CH, the practical is fairly small. Furthermore, the CH shortest path queries are much faster as they only need to build two upwards trees[2] in a Dijkstra manner and are thus relaxing far fewer edges compared to a normal Dijkstra query.

[2]i.e. recursively following upwards edges from one node

Using the CH as an improvement, we change the representation of the $s \in \mathcal{S}$ from node paths to CH paths. To achieve this, we either have to compute the minimal infeasible paths directly as CH paths or we can use the information obtained by the calculation of the node paths, to get the corresponding CH paths. As CH shortest path calculations are extremely fast and the direct construction of the minimal infeasible CH paths is elaborate, we choose the naive way of first computing the node paths and then the CH paths afterwards.

This is done by storing the leaves of the minimal infeasible paths from the normal Dijkstra tree, for each source node $v \in V$. Then we compute CH Dijkstras from v to each of its leaf nodes, to obtain the edges of the corresponding CH path. Note that we ignored the fact, that we can not hit the first or the last node of a minimal infeasible path. This can be taken into account by either shortening the node path or by adapting the hitting set algorithm. Either way, this problem is of rather technical nature and thus ignored in favor of a high level description.

Algorithm 4.1 The CH BLS positioning algorithm

Input:$G = (V, E)$, M **Output:**$\mathcal{H}_g$

1: $\mathcal{S}_M = \emptyset$
2: $\mathcal{H}_g = \emptyset$
3:
4: **for all** $v \in V$ **do** // Calculate the SPSS
5: $\quad P =$ minimal infeasible paths, that start in v with initial battery charge M
6: $\quad P_{CH} =$ corresponding CH paths of P
7: $\quad \mathcal{S}_M = \mathcal{S}_M \cup P_{CH}$
8: **end for**
9: // Calculate the hitting set
10: **while** $\mathcal{S}_M \neq \emptyset$ **do**
11: $\quad$ count the occurrences of each $e \in E$ in $\mathcal{S}_M$
12: $\quad$ convert the edge counts to node counts
13: $\quad v_{max} =$ node with the most occurrences in $\mathcal{S}_M$
14: $\quad \mathcal{H}_g = \mathcal{H}_g \cup \{v_{max}\}$
15: $\quad E_{max} =$ all edges v_{max} is implicitly or explicitly a part of
16: $\quad \mathcal{S}_M = \mathcal{S}_M \setminus \{s \in \mathcal{S}_M | \exists e \in E_{max} : e \in s\}$
17: **end while**
18:
19: **return** $\mathcal{H}_g$ // Return the computed hitting set

Algorithm 4.1 shows the resulting algorithm. Changes to the old algorithm are marked in red. The important changes are located in the lines 6, 12 and 15. The naive conversion of a CH path to a normal path, that is needed in line 6, is described in the last two paragraphs. In line 12 we want to transfer the edge counts to node counts. A property of the CH comes to help. We can order the edges descending, according to the time they have been created in the CH preprocessing. Then we pass over the edges and transfer the counts to the child edges in case of a shortcut, or to the source and target nodes in case of a normal edge. Metaphorically speaking we're *unfolding* the CH, while correctly maintaining the counts. Then, as $\mathcal{S}_M$ most likely contains shortcuts, we have to somehow *fold* the CH to delete the hit sets from it. This

occurs in line 15, where we take all adjacent edges of v_{max} and recursively take their parent edges to obtain all the edges that implicitly or explicitly contain v_{max}.

While this new algorithm will reduce the memory usage drastically (especially when using large graphs), the runtime should not change notably in practice. Line 6 just adds the time of a CH shortest path query, after a standard Dijkstra query has already been accomplished in the last line. As a CH shortest path query is faster than a normal Dijkstra by some orders of magnitude, this will not affect the runtime. The calculations in line 12 and 15 both run in $O(|E|)$ and can be safely ignored concerning the runtime as $|\mathcal{S}_M| \gg |E|$ in practice and a pass over $\mathcal{S}_M$ is necessary in line 16.

4.2 Runtime reduction

4.2.1 Incremental hitting set construction

In the first section of this chapter we only reduced the memory usage of the set system but not the calculation time. Still, when the battery capacity increases and the graph is large enough, the set system calculation time increases as well. Finally it converges to

$$|V| \cdot |\text{average time of a one-to-all Dijkstra}|$$

as we would perform full one-to-all standard Dijkstras for each node. For an 18 million nodes graph with a Dijkstra calculation time on a plain graph of over 2 seconds, this would mean a single threaded runtime of over a year[3]. Remember that it is hard to apply normal shortest path calculation speedups in the EV setting and we thus have to use the calculation time on a plain graph. It should be unnecessary to mention that this is impractical and has to be improved, as we aim for practicability.

A promising idea is to divide the set system creation into subproblems, where we do not need to perform Dijkstra calculations with a large battery capacity for many nodes. And indeed, this is possible by first constructing a hitting set for a smaller battery capacity M_1, and then using this first hitting set $\mathcal{H}(\mathcal{S}_{M_1})$ to construct the final hitting set, with respect to the battery capacity M. This is done by first partitioning M into $M = M_1 + M_2$. Then we construct the hitting set for the battery capacity M_1 and plug the result into the calculation of M_2, by only allowing nodes in $\mathcal{H}(\mathcal{S}_{M_1})$ to be source nodes of the paths in the set system. We call the resulting set system $\mathcal{S}_{M_1,M_2}$. Note that this notation includes an implicit hitting set calculation for M_1. Of course this approach can also be used with multiple rounds, creating a set system $\mathcal{S}_{M_1,M_2,\dots,M_k}$, where k is the number of rounds:

Definition ($\mathcal{S}_{M_1,M_2,\dots,M_k}$)**.** $\mathcal{S}_{M_1,M_2,\dots,M_k}$ *is the SPSS that is constructed by recursively calculating* $\mathcal{S}_{M_1,\dots,M_i}$*, allowing only nodes from* $\mathcal{H}(\mathcal{S}_{M_1,\dots,M_{i-1}})$ *to be source nodes of its paths, with* V *being the set of source nodes for* $\mathcal{S}_{M_1}$*.*

[3]We took the graph size and the one-to-all standard Dijkstra time (which they mentioned to be hard to improve) from [DGNW12].

Note that $\mathcal{H}$ denotes an arbitrary hitting set, without specifying it being minimum or greedily constructed . If the distinction is of importance, it is noted in the following.

Obviously, this approach can accelerate the calculation when we choose the right parameters. The idea is to first reduce the number of nodes, so that we do not have to construct many large Dijkstra trees at the end. To achieve this with little runtime, we choose small M_i at the beginning. When we reach a satisfiable amount of reduction, we're aiming at computing a final hitting set of good quality, which is possible by choosing a big M_k. The last step should not take much time because of the previous steps.

In the following, proofs are given for the correctness of the incremental approach.

Claim. *$\mathcal{H}(\mathcal{S}_{M_1,M_2})$ is a hitting set for $\mathcal{S}_M$, with $M = M_1 + M_2$.*

Proof. For every arbitrary path $s \in \mathcal{S}_M$, we show that it must be hit by a node from $\mathcal{H}(\mathcal{S}_{M_1,M_2})$. We know that

1. every shortest path starting at a node $v \in V$ either ends in a dead end or is hit by a node $v' \in \mathcal{H}(\mathcal{S}_{M_1})$, without the battery running empty and the initial and maximum charge being M_1.
2. every shortest path from a node $v' \in \mathcal{H}(\mathcal{S}_{M_1})$ either ends in a dead end or is hit by a node $v'' \in \mathcal{H}(\mathcal{S}_{M_1,M_2})$, without the battery running empty and the initial and maximum charge being M_2.

It follows that

$$\forall p \in \mathcal{S}_M \exists v' \in \mathcal{H}(\mathcal{S}_{M_1}), v'' \in \mathcal{H}(\mathcal{S}_{M_1,M_2}) : p \text{ corresponds to a path } v \xrightarrow{M_1} v' \xrightarrow{M_2} v''$$

where $x \xrightarrow{M} y$ means that we can reach y from x with an initial and maximum battery charge of M. Therefore, p can be driven with an initial and maximum battery charge $M_1 + M_2 = M$. □

The claim also holds with more incremental steps:

Claim. *$\mathcal{H}(\mathcal{S}_{M_1,\dots,M_k})$ is a hitting set for $\mathcal{S}_M$, with $M = M_1 + \dots + M_k$ and $k \in \mathbb{N}^+$.*

Proof. From the last proof we know that $\mathcal{H}(\mathcal{S}_{M_1,M_2})$ is a hitting set for $\mathcal{S}_{M_1+M_2}$. Thus, also every $\mathcal{H}(\mathcal{S}_{M_1,M_2,\dots,M_k})$ is a hitting set for $\mathcal{S}_{M_1+M_2,M_3,\dots,M_k}$. The claim follows by induction. □

If we have edges with higher battery consumption than M that are part of a shortest path, then we have to stop the computation and return an error as we can not take these edges without the battery getting empty. Thus, the problem is unsolvable. However, as the minimal possible M increases, the running time of our incremental algorithm might increase as well. Therefore we're interested in keeping M small. We can solve this problem by pre-hitting the nodes, that are the source of an edge longer than M, which also is part of a shortest path. Note that this still is not a solution of the hitting set problem for the current M, but we can now use the created hitting set as an input for the next step in an incremental construction.

In the following we give a proof that the pre-hitting does not affect the correctness of an incremental hitting set construction, in case the final step does not apply pre-hitting.

Claim. *$\mathcal{H}(\mathcal{S}_{M_1,\ldots,M_k})$ is a hitting set for $\mathcal{S}_M$, with $M = M_1 + \ldots + M_k$ and where $\mathcal{H}(\mathcal{S}_{M_1}), \mathcal{H}(\mathcal{S}_{M_1,M_2}), \ldots, \mathcal{H}(\mathcal{S}_{M_1,\ldots,M_{k-1}})$ were possibly calculated using pre-hitting.*

Proof. We have to prove that the pre-hitting modification does not affect the correctness, as long as a non pre-hitting step follows. In every step x before the non pre-hitting one, we always implicitly hit the shortest path prefix with respect to a battery charge $\sum_{i=1}^{x} M_i$. Except for the edges $e \in E$ with $\eta(e) > M_x$, where we could not build a prefix path. By hitting the source nodes of these edges, we assure that in the step y, where $\eta(e) \leq M_y$, the shortest paths starting with one of these edges are hit, with respect to an initial and maximum battery charge of M_y. It follows, that also every path with respect to an initial and maximum battery charge of M is hit because $M_y \leq M$. □

4.2.2 Multiple hitters

Especially for small M, the hitting set $\mathcal{H}(\mathcal{S}_M)$ is rather large and therefore the loop beginning in line 10 in algorithm 4.1 is passed through many times, because we always only take one hitter per round. This can be improved by the multiple hitters approach, where we take k instead of one node per round into the hitting set. This technique is based on the assumption that two hitters, that are in two parts of the graph that are a certain distance d apart, hardly affect each other in the greedy algorithm.

In Figure 4.2, one can see an example of the partial independence of subgraphs in the multiple hitters setting. The numbers next to the nodes are the number of paths from the set system, that run over this particular node. For simplicity we assume in this example that the EV battery charge lasts for two edges and on the third it runs empty. This means that all the minimum infeasible paths consist of three edges; thus, four nodes. We can not hit the first and the last node as explained in 3.1, and therefore only the two nodes in the middle of each minimum infeasible path can be hit. Furthermore, we consider the paths to be undirected.

At the beginning of the hitting set algorithm there are two nodes with the maximum value '12'. As the graph is symmetric it does not matter which node we hit first, so we take the left one. As shown in the figure, the counts of five nodes on the left side of the graph change, while the counts on the right side remain the same. In the next step we would hit the other node with count '12'. Because of the partial independence of the left and the right side, we also could have hit both maximum nodes at the same time and the result would not have changed.

It is hard to evaluate which hitters can be taken into the hitting set independently and therefore we rely on a heuristic. As already mentioned, we assume that a certain distance between two hitters guarantees that they hardly affect each other. So when we choose hitter number x, we have to check if it is far enough away from the last $x - 1$ hitters. This can be done by a simple check of the pairwise euclidean distance. Though this process of taking k hitters is in $O(k^2)$ assumed we only have to check a constant number of nodes, it is still much faster than

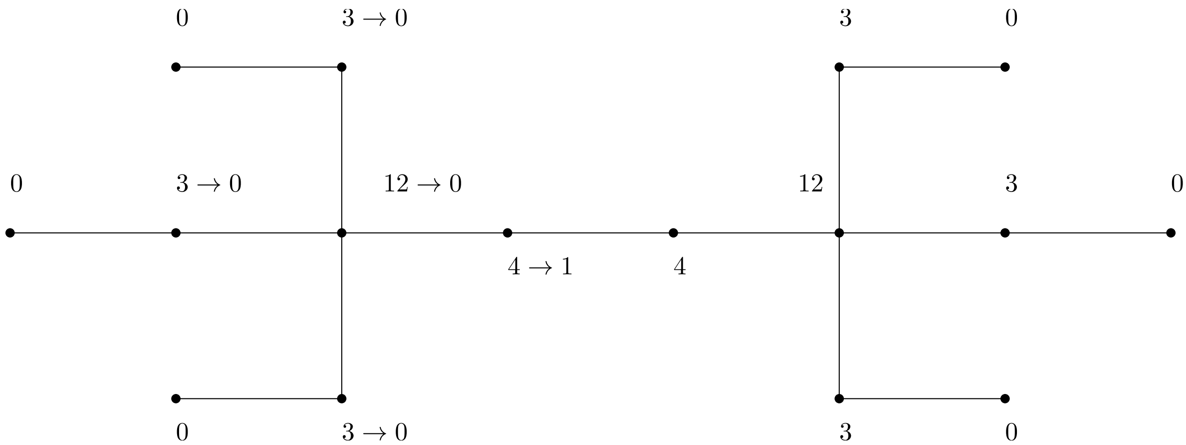


Figure 4.2: Example of the partial independence of greedy hitting.

the former approach, as long as k is not too big. However, we still deal with an approximation and thus check the performance in practice in chapter 5.

To apply the multiple hitters strategy to algorithm 4.1, we have to change lines 13 to 15 to the lines in algorithm 4.2 and add an additional input parameter k, that denominates the number of hitters per round.

Algorithm 4.2 Multiple hitters adaption

Input: k

$V_{max} = k$ nodes with the most occurrences in $\mathcal{S}_M$ that are far enough apart
$\mathcal{H}_g = \mathcal{H}_g \cup V_{max}$
$E_{max} =$ all edges one node from V_{max} is implicitly or explicitly a part of

---

4.2.3 Parallelization

The last improvement is a technical one. Especially with current hardware it has become common to use multiple CPUs, to support better multithreading. If a task can be divided into several subtasks, a notable speedup is possible. As our testing machine has 24 cores, the parallelization of the algorithm is very promising. Also, the division into subtasks is possible in most parts. Algorithm 4.1 reflects the steps we have to take to compute a hitting set, while small changes are omitted that were introduced with the multiple hitters strategy in section 4.2.2. Because these changes do not affect the parallel execution, we take this algorithm to analyze the possibilities of a speedup by parallelization.

Lines 4 to 8 can be trivially parallelized by partitioning the nodes, so that every thread computes his own set system. In the hitting set computation we use the fact that we partitioned the nodes, and therefore also the set system, to reduce conflicts while running in parallel. This is done in lines 11, 13 and 16. In line 11 we want to count the number of times each edge occurs in $\mathcal{S}_M$. As the set system is partitioned, we let every thread count its own part and then merge

the counts. In line 13 we search for the maximum in an array of counts. By partitioning the array, so that every thread gets an equally big part, we can determine τ local maxima, where τ is the number of threads. From these local maxima, we can easily decide a global maximum. Then, in line 16, the partitioning of $\mathcal{S}_M$ helps again, because every thread can independently delete the sets in its own part of the set system.

In contrary to the lines mentioned in the last paragraph, it is elaborate to parallelize lines 12 and 15. In these lines we either *fold* or *unfold* CH edges. Fortunately, these tasks are not that time consuming. Therefore, in practice no problems arise, when we run these parts single threaded.

CHAPTER 5

Analysis

For our final implementation, we used the naive path extraction combined with the CH representation of the paths in $\mathcal{S}$ and a multiple hitters strategy, to position the BLSs. The graph data we used for the tests was taken from the OpenStreetMap project[1]. We use five different subgraphs of Germany for our tests: Pforzheim ($|V| = 150k$, $|E| = 305k$), the administrative region of Tübingen ($|V| = 500k$, $|E| = 1mio$), the southern part of Baden-Württemberg ($|V| = 2.2mio$, $|E| = 4.6mio$), southern Germany ($|V| = 4.2mio$, $|E| = 8.6mio$) and Germany ($|V| = 17.7mio$, $|E| = 36mio$). For the two smallest graphs Pforzheim and Tübingen, we set the battery capacity $M = 40k$ and for the others $M = 125k$, when not mentioned otherwise. The values of M correspond to the maximum distance in meters that can be driven on flat terrain, without the battery running empty. Our C++ code was evaluated on a 24 core machine with two AMD Opteron 6172 CPUs at 2.1GHz and 96GB of RAM.

A detailed overview of the performance of our implemented version of the BLS positioning can be seen in Table 5.1. For this tests, we only used the calculation in two steps. Nevertheless, more steps can speedup the calculation even further without worsening the quality significantly. However, as there are numerous possibilities for partitioning M, even with a fixed number of steps, we do not extend our detailed tests to more than two steps. Additionally, we implemented a naive path packing algorithm to obtain a lower bound of the hitting set size, and thereby an approximation ratio, as mentioned in section 2.2.3. The tests show that it takes considerable more time to build $\mathcal{S}$ than to calculate the corresponding hitting set. This is partly because of the multiple hitters strategy deployed in the greedy hitting set calculation. By modifying this strategy, while still assuring that the quality does not drop, even further speedups are possible but not urgent in our case. So, as expected, the path system creation is the crucial point.

[1] http://www.osm.org

#nodes (M_1,M_2)	CPU times 1. iteration		CPU times 2. iteration		overall time		$\|\mathcal{H}\|$	APX
	path system	hitting set	path system	hitting set	CPU	real		
150k (2k,38k)	141s	16s	396s	10s	563s	34s	29	2.90
500k (2k,38k)	736s	107s	1,388s	168s	2,399s	171s	180	3.27
2.2mio (2k,123k)	10,262s	324s	113,390s	488s	124,464s	5,451s	94	3.03
4.2mio (2k,123k)	28,931s	2,063s	182,730s	4,453s	218,177s	9,648s	180	3.00
17.7mio (1k,124k)	670,844s	338,590s	4,686,968s	57,733s	5,754,135s	302,822s	1069	3.39

Overview of the test results of the incremental implementation (2 steps) with a CH path system and a multiple hitters speedup. The implementation was parallelized; thus the CPU time and the real time differ strongly. We did not run into memory shortage in the incremental cases.

Table 5.1: Overview of the test results of the final implementation.

#nodes	Naive representation		CH representation		incr. 2 steps		incr. 5 steps	
	time	#elements	time	#elements	time	inc. of $\|\mathcal{H}\|$	time	inc. of $\|\mathcal{H}\|$
150k	11,707s	$13.2 \cdot 10^9$	13,785s	$0.2 \cdot 10^9$	563s	1.03	478s	1.07
500k	-	-	72,299s	$1.3 \cdot 10^9$	2,399s	1.04	3,867s	1.07

Timings (CPU time), sizes of the set systems, penalties for incremental construction. We assumed a battery capacity which allowed traveling for about 40km on flat terrain. The Tübingen graph with 500k nodes could not be processed with the naive representation.

Table 5.2: Comparison of the test results of different approaches.

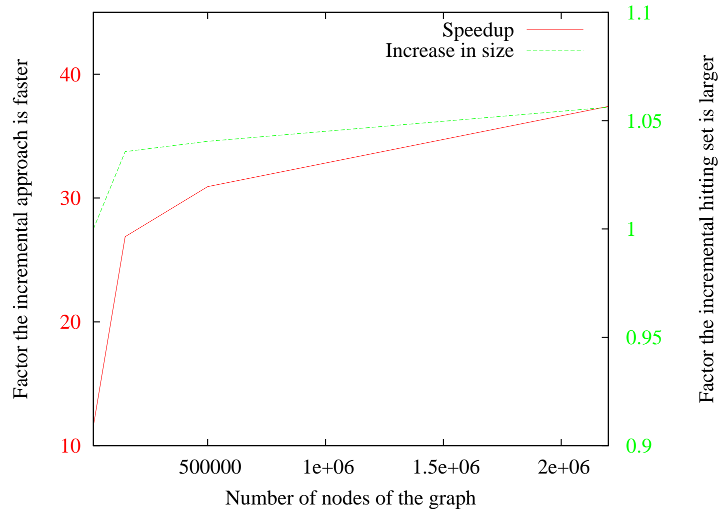
5.1 Memory usage

In chapter 4, we first introduced the improvement of the set system data structure to reduce the memory usage. In theory the memory usage can be reduced by a factor of 5. In practice we could measure reductions by the factor of 3, when computing set systems for a small value of M. When M increases, the memory reduction factor decreases. This corresponds directly to the theoretical analysis. Especially, when we compute the set system incrementally, the memory reduction is significant and brings great practical improvements.

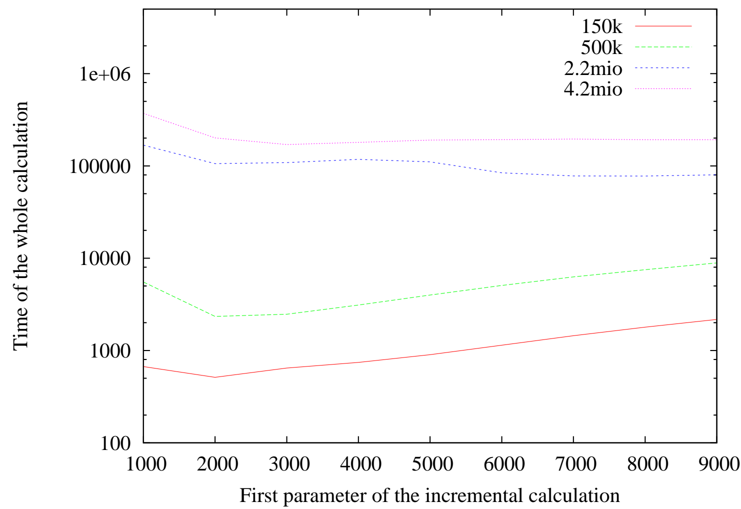
The results shown in Table 5.2, where we compare our final implementation with the non-CH and non-incremental approaches, clearly demonstrate its superiority. Without the CH representation, the non-incremental calculation of our test on the $500k$-graph is not even possible as we run out of memory. Also, the set system is two orders of magnitudes larger for the $150k$-graph. Thus, if we want to apply our BLS positioning in practice, we can not rely on the naive path representation and should use the CH representation instead.

5.2 Calculation time

While we do not run into memory issues anymore when changing the path representation and improving the set system structure, the computation time would still prevent our implementation from being applicable to larger graphs in practice. Therefore, we chose to add the incremental approach to allow for a considerable speedup. The downside of this approach is the



(a) Speedup by incremental construction.



(b) The plot shows the overall time of incremental calculations with different partitioning of M. The X-axis shows the value of M_1; thus, the value of M_2 is implicitly given. The graphs are identified with respect to their number of nodes.

Figure 5.1: Visualization of the performance of the incremental approach

possible increase in size of the final hitting set, lowering the quality of the result. Fortunately, the size almost remains the same as the results in the columns 'inc. of $|\mathcal{H}|$' of Table 5.2 show. In Figure 5.1a, we compare the speedup to the quality loss on graphs with different size:

The maximum growth of the size of the incremental hitting set is approximately 1.05, which can be seen as insignificant. In contrary, the calculation time is reduced by more than one order of magnitude. With parameters that are chosen more carefully these results can be improved even further. In Figure 5.1b, one can see how the incremental parameters affect the overall calculation time.

Another speedup we introduced in chapter 4 is the multiple hitters approach, where we take more than one hitter per round. Again, there has to be a tradeoff between the size of the hitting set and the acceleration of the calculation. Furthermore, there are two parameters that can be chosen: the number of hitters per round k and the minimum euclidean distance d two hitters must exhibit to be hit together in one round. Thus, we tested combinations of these two parameters on the graph of Pforzheim ($|V| = 150k$) and an M that implies an average reach of one kilometer: the results can be seen in Table 5.3. The obtained values show that choosing the right parameters is a very delicate process as there are many side effects involved. Especially, d has an effect on the number of hitters we can take into the hitting set per round, because there might not exist k nodes that are as far apart as d. This side effect explains the similar values in the results for $d = 32M$ for different k. When we increase the minimum distance d, there is no monotonic increase or decrease in the time and the number of hitters. The results for very small values and very large values of d are rather bad. By trend, the results for $d = M$ are the best ones, which can be explained with the fact that this should be close to the average distance of a path in the set system and thus, the already hit paths of this round are unlikely to be hit again. Though the long calculation time for $d = 32M$ was expected, the higher number of hitters is rather unexpected. This might occur because of the unnaturally chosen values of d, which leads to an exclusion of many useful hitters in favor of worse hitters that lay just outside the boundary of the chosen euclidean distance. When choosing the number of hitters per round, there is a classical tradeoff between quality and calculation time. By trend, the more hitters we settle per round, the worse the quality gets and the faster the calculation is. This effect is reduced by increasing the value of d. However, when the value exceeds a certain level, the overall effects on the calculation worsen as already described. The huge sizes of the hitting set, when choosing $k = 1024$, suggests that we should tend to choose k small; especially, when we compute the final hitting set in the incremental approach.

Results for the effects of the parallelization can also be read from the already collected data. In Table 5.1 one can see the measured overall time for the different calculations; on the one side the real time and on the other the CPU time. From this we can derive an approximation of the speedup obtained by the parallelization. The ratio of the CPU time and the real time lies between 14 and 22.8 while, as already mentioned, the maximum speedup is a factor of 24 as we used a machine with that many CPUs.

k \ d	0	$\frac{M}{2}$	M	$2M$	$4M$	$32M$
1	1523.13s / 4351	1524.26s / 4351	1528.14s / 4351	1525.20s / 4351	1524.08s / 4351	1520.63s / 4351
2	772.80s / 4428	763.13s / 4368	763.15s / 4362	762.33s / 4362	760.73s / 4358	927.82s / 5076
4	406.52s / 4618	384.08s / 4382	384.84s / 4382	386.05s / 4386	385.82s / 4386	941.86s / 5863
16	115.88s / 5234	99.20s / 4450	98.76s / 4434	99.82s / 4450	102.05s / 4546	940.27s / 5863
64	41.91s / 7426	26.96s / 4610	26.94s / 4610	33.34s / 5634	72.73s / 12610	940.56s / 5863
1024	10.20s / 24578	9.19s / 17410	52.47s / 24946	65.48s / 19761	97.66s / 14531	936.79s / 5863

Multiple hitters test, where k is the number of hitters per round and d is the distance between two hitters expressed in M, which is the reach of the EV on flat terrain. The structure of the entries is: '<calculation time of the hitting set> / <size of the resulting hitting set>'.

Table 5.3: Results of multiple hitters tests.

5.3 Approximation bounds

Observing the algorithm, one can see that only in the hitting set part approximation is used. The set system is calculated properly. So the question arises how good the approximation of the hitting set is in theory and practice; particularly, under the aspect that we are approximating on several parts of the hitting set algorithm.. This could worsen the result as the approximation errors might amplify each other.

The points where we approximate are:

- the hitting set in general by a greedy algorithm
- when we construct the hitting set incrementally
- when we take multiple hitters in one round

We analyze all the above approximations and derive some bounds, concerning theory and practice.

5.3.1 Theoretical bounds

For the hitting set calculation we use the greedy algorithm, where we always choose the node or one of the nodes contained in the most number of sets. This algorithm is a $\log n$-approximation [Chv79].

Then, we also construct the hitting set incrementally as described in 4.2.1. We are interested in an upper bound of the size of an incremental hitting set $\mathcal{H}_g(\mathcal{S}_{M_1,\ldots,M_k})$ and we assume that the incremental hitting set will always be constructed in a greedy manner. Finding an upper bound is not that easy because the following holds:

Claim. *There exists no universal upper bound for the size of an incremental hitting set $\mathcal{H}_g(\mathcal{S}_{M_1,\ldots,M_k})$, which can be derived from the size of the non incremental hitting set $\mathcal{H}_g(\mathcal{S}_M)$ with $M = \sum_{i=1}^{k} M_i$.*

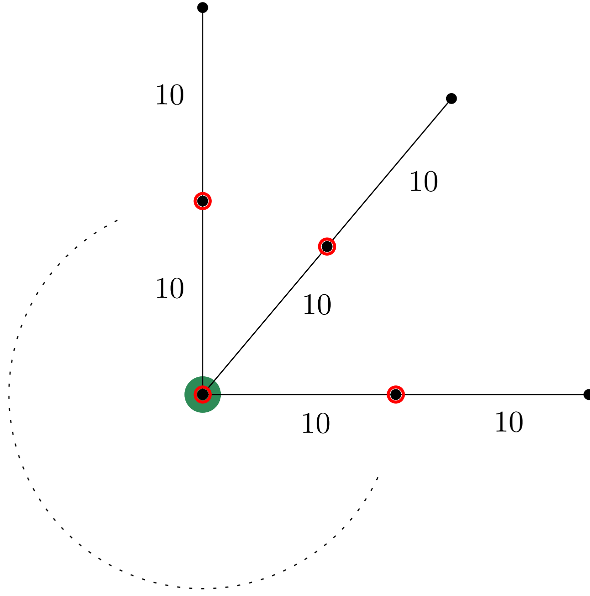


Figure 5.2: There is no upper bound of the incremental hitting set, using $|\mathcal{H}(\mathcal{S}_M)|$.

Proof. The claim is shown by defining a graph $G = (V, E)$, a battery capacity M and a partitioning of M into M_1 and M_2, where no such upper bound is possible. For the sake of simplicity we visualize the example in Figure 5.2. The graph is also implicitly given by the visualization, where we do not specify the exact number of 'arms' it has. Furthermore, we choose $M = 22$ and $M_1 = M_2 = 11$.

On one hand, the greedy hitting set for $\mathcal{S}_M$ is marked in green and it is unambiguous, because every infeasible path of length 22 has to contain the center node, whereas for each other node there exists a minimal infeasible path in which it is not contained. Thus, $|\mathcal{H}_g(\mathcal{S}_M)| = 1$. On the other hand, the greedy hitting set for $\mathcal{S}_{M_1,M_2}$ is marked in red and it is also unambiguous. This is the case as all paths in the set system $\mathcal{S}_{M_1}$ contain exactly one node to be hit, because there are only edges with battery consumption 10 and after one edge, the EV runs out of energy. All these nodes have to be hit and the result is the set of nodes marked in red. Though, the set system $\mathcal{S}_{M_1,M_2}$ is restricted to source nodes of $\mathcal{H}_g(\mathcal{S}_{M_1})$, the hitting set does not change.

As we can choose the number of 'arms' of the graph to be arbitrarily large and as the number of hitters for the incremental case increases with the number of 'arms', there exists no upper bound for

$$\frac{|\mathcal{H}_g(\mathcal{S}_{M_1,M_2})|}{|\mathcal{H}_g(\mathcal{S}_M)|}$$

and thus also no universal upper bound for hitting sets $\mathcal{H}_g(\mathcal{S}_{M_1,\dots,M_k})$ depending on the size of $\mathcal{H}_g(\mathcal{S}_M)$. □

Note that it does not matter if we compute the greedy or the minimum hitting sets in the previous proof. Instead of proving an upper bound with the size of the hitting set $\mathcal{H}(\mathcal{S}_M)$, $M = \sum_{i=1}^{k} M_i$, we prove a bound with the size of $\mathcal{H}(\mathcal{S}_{M_k})$:

Claim. $|\mathcal{H}_g(\mathcal{S}_{M_1,\ldots,M_k})| \leq log(n) \cdot |\mathcal{H}_{min}(S_{M_k})|$

Expressed in words, the claim states that the size of a *greedy* hitting set $\mathcal{H}_g(\mathcal{S}_{M_1,\ldots,M_k})$ of the *incremental* set system $\mathcal{S}_{M_1,\ldots,M_k}$ is bounded by the size of a *minimum* hitting set $\mathcal{H}_{min}(\mathcal{S}_{M_k})$ of the *non incremental* set system $\mathcal{S}_{M_k}$ (which only uses the last battery capacity M_k of the incremental construction), multiplied by $\log(n)$. Following from this, the theoretical results recommend choosing a large M_k to keep $|\mathcal{H}_{min}(S_{M_k})|$ small and, thus, forcing a low upper bound for the size of the incremental hitting set.

Proof. From the fact that a greedy hitting set is a $\log n$-approximation, we know that

$$|\mathcal{H}_g(\mathcal{S}_{M_1,\ldots,M_k})| \leq \log(n) \cdot |\mathcal{H}_{min}(\mathcal{S}_{M_1,\ldots,M_k})|$$

where $\mathcal{H}_{min}(\mathcal{S}_{M_1,\ldots,M_k})$ means that only the hitting set for M_k has to be a minimum hitting set while those for M_1 to M_{k-1} could be approximated. Also

$$|\mathcal{H}_{min}(\mathcal{S}_{M_1,\ldots,M_k})| \leq |\mathcal{H}_{min}(\mathcal{S}_{M_k})|$$

holds, following from the fact that $\mathcal{S}_{M_1,\ldots,M_k} \subseteq \mathcal{S}_{M_k}$ which itself holds because of $\mathcal{H}(\mathcal{S}_{M_1,\ldots,M_{k-1}}) \subseteq V$ and thus

$$\log(n) \cdot |\mathcal{H}_{min}(\mathcal{S}_{M_1,\ldots,M_k})| \leq \log(n) \cdot |\mathcal{H}_{min}(\mathcal{S}_{M_k})| \,.$$

Therefore, the upper bound of the claim holds. □

It is also possible to derive upper bounds for the multiple hitters approach introduced in 4.2.2. As already explained, there must be a certain euclidean distance between those k nodes we hit in one round. However, the distance is not related to the battery usage it takes to get from one node, out of these k, to another one. Assumed that we can get from the first node to each other node, still (or again) having a full battery when arriving, then all the other nodes are useless hitters. It follows that the hitting set contains k times as many nodes as when we would only pick the most frequent node in every round. The latter is a greedy strategy and thus a $\log n$-approximation. All in all, the multiple hitters strategy is thus a $k \cdot \log n$-approximation.

These results show that in theory our algorithm is condemned to failure. Fortunately, all these worst-cases of the theoretical bounds do not emerge when we work on a graph of a road network.

5.3.2 Instance-based bounds

To obtain an instance-based bound for a hitting set $\mathcal{H}(\mathcal{S})$, we can compute a set packing $\mathcal{P} \subseteq \mathcal{S}$. Its size $|\mathcal{P}|$ forms a lower bound as explained in section 2.2. Thus, the instance-based approximation ratio is given by $\frac{|\mathcal{H}(\mathcal{S})|}{|\mathcal{P}|}$. As we already need to accelerate the computation of the hitting set, it seems unlikely that the set packing computation is practical for bigger graphs, when implemented in a naive way.

Therefore, the approximation values in Table 5.1 were determined by a similar but faster technique that can be seen in algorithm 5.1. Instead of building an explicit set system and a corresponding set packing, we directly build a path packing. This is done by randomly picking nodes that we start a Dijkstra tree calculation on with initial battery charge M. Then we extract the minimal ev-infeasible paths and put those into $\mathcal{P}$, which do not overlap with one of the other paths in $\mathcal{P}$. We stop when $|\mathcal{P}|$ does not grow for a certain amount of rounds. The whole procedure is repeated as long as desired. This round based approach combined with the randomization reduces the probability of finding only local maxima that are far away from the global one.

Algorithm 5.1 Practical packing algorithm

Input:$G = (V, E)$, M **Output:**the maximal $|\mathcal{P}|$

```
1:  maximum = 0
2:  while there is still some time left do
3:      P = ∅
4:      while still something changes do
5:          v = random node from V
6:          P = minimal infeasible paths, that start in v with initial battery charge M
7:          P = P ∪ {p ∈ P|∀q ∈ P : p ∩ q = ∅}
8:      end while
9:      maximum = max{maximum, |P|}
10: end while
11: return maximum
```

5.4 Quality

There are several ways to measure the quality of the hitting sets we compute. We can evaluate it depending on:

- the size of the resulting hitting set
- the approximation ratio we obtain via set packing
- manual visual analysis of the hitters, placed on the corresponding map

We already computed several hitting set sizes and approximation ratios which can be seen in Table 5.1. Naturally, these two values correlate strongly. However, while the approximation ratio measures the quality of our algorithm, the overall size gives a hint about the quality of the approach we chose (i.e. hitting every shortest path concerning a certain battery capacity M).

In particular, we can relate the hitting set size to the actual number of gas stations in the same area we used for the calculation. On one side, our calculation on the graph of Germany results in a hitting set size of $1,069$ with an average reach of $125km$ as can be seen in Table 5.1. On the other side, at the beginning of the year 2013 there existed about $14,300$ gas stations in Germany [Minb]. However, note that we only computed BLSs with the aim to hit all long distance paths, ignoring the stations that need to be placed for short distances in practice; as for example on routes leading to workplaces or supermarkets. On the contrary, our BLSs were placed on directed paths which might increase their count unnecessarily. A better comparison is the number of gas stations placed at highways. In Germany they sum up to approximately 350[2]. When considering the higher range of 'normal fueled cars', our positioning seems to be excellent concerning the number of hitters.

The approximation ratio also suggests our results to be good ones, as in Table 5.1 it does not raise above 3.4, even on the road graph of Germany. Note that the instance-based bounds are approximated themselves and could be lower if we improve the packing algorithm.

As we use real road graphs for our calculations, a quality check is possible as well by connecting the hitting set data with the map data on which it has been computed. For example, we can test which road types are hit, whether crossroads are preferred, how well distributed the hitting set is and what effects the 'directed hitting' has on the results. Figure 5.3 shows an extract of a computed BLS positioning on Germany with a battery capacity that allows for a drive of approximately $125km$ of distance on flat terrain. The specific section of the map shows a part of northeastern Germany. Naturally the positioning prefers road types that allow for a higher speed and a bigger capacity. Only 2 out of 17 BLSs are placed on road types smaller than the biggest two (the 'Autobahn' in blue and the 'Bundesstraße' in red) shown in the picture. Additionally, the one to the left seems to be chosen because of faulty graph data, as the shortest paths through the city are most likely not the fastest in practice. Moving to another aspect, the hitting of crossroads, we can see that not as many crossroads as expected are hit. One reason for this might be that we're hitting directed paths and if the graph represents the two driving lanes of a street as two different paths then the importance of hitting crossroads is reduced and the importance of big road types prevails. Another aspect that is visualized very good by the chosen extract is the distribution of the BLSs. It varies strongly depending on the concentration of the various road types in the respective parts. If the concentration is balanced, then also the distribution of the BLSs is balanced, which can be seen in the lower left of the picture. If, however, the opposite is the case, as can be seen on the 'Autobahn' that crosses the map, then the positioning prefers the biggest roads and, thus, becomes unbalanced. Note that this is a desired and also expected behavior. Effects of the fact that the set system consists of directed paths can be seen in all parts of the picture. Especially

[2] This can be calculated out of the values from [Minb] and [Mina]

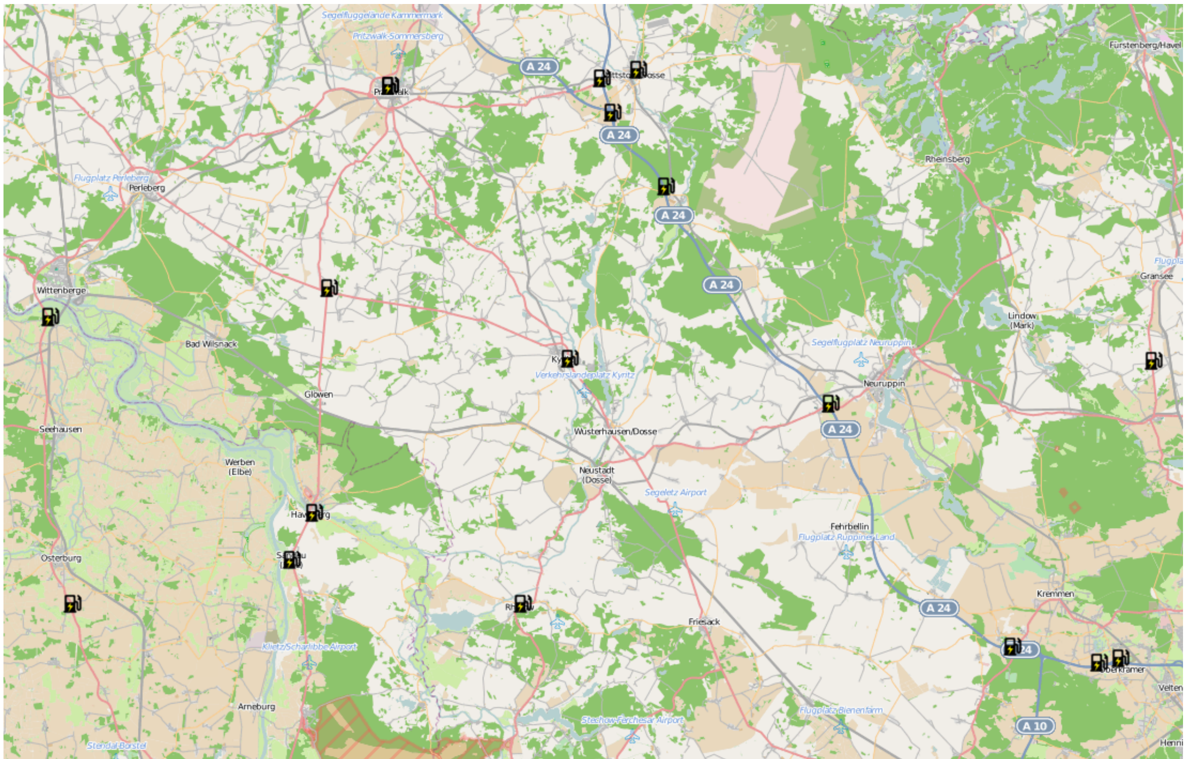


Figure 5.3: An example of a computed BLS positioning.

the 'Autobahn' contains many BLSs that are concentrated on a short segment which would be very unlikely if we do not use directed paths. This behavior exhibits a possible improvement of the algorithm by identifying these close hitters that arise from the usage of directed paths.

CHAPTER 6

Conclusion and future work

The aim of the thesis is to place battery loading stations (BLSs) such that every shortest path can be driven by an electric vehicle (EV) without running out of energy. Furthermore, this network of stations should be computable in acceptable time and with a reasonable amount of memory. To minimize the costs of an initial roll-out, we want to place as few stations as possible, given the previous mentioned constraints.

We first introduced into the topic of EV routing and defined it formally by giving functions for the calculation of the battery consumption for a given edge and the determination of ev-feasibility. Ev-feasibility lays the foundation for our problem by defining for given paths if they can be driven with an EV, without the battery getting empty. This helps us defining the paths that we need to place BLSs on. The battery loading stations themselves are represented as nodes where the battery charge is recuperated completely. We then introduced the formal fundamentals that are necessary to understand the following chapters in detail. Namely, we had a look at set systems, hitting sets, set packings and contraction hierarchies (CHs).

Having introduced all the necessary topics, we then finally presented the first naive algorithm for the problem of placing BLSs. The algorithm can be divided into two parts. The first part is the creation of a set system of all shortest paths we need to place a BLS on, excluding obviously dependent sets. The second part is the computation of a hitting set for the previously calculated set system. By hitting all the necessary paths and thus placing required BLSs, we have met the requirement of making every shortest path drivable with an EV.

As the implementation of the naive approach did not meet our requirements concerning computation time and memory usage, we applied several improvements to establish practicability. We first reduced the memory usage by improving the set system structure used in the implementation; replacing list and array classes by one big C array and reducing the

memory overhead close to the minimal possible. To further reduce the memory usage of the set system, we changed the path representation from the naive node representation to the representation by CH paths because they contain much less edges, especially when we deal with long paths. A technique that reduces both, the memory usage and the computation time is the incremental construction of the set system. It divides the given problem into smaller subproblems of the same type, where the calculation time of all the subproblems does not add up to the original one. As the subproblems construct their own set systems, which are smaller naturally, the space consumption is reduced as well and equals the maximum memory usage of a subproblem. Aiming at a further reduction of the calculation time, we introduced the multiple hitters technique by extending the greedy hitting set algorithm by a procedure that chooses several hitters per round, where normally only the best node is taken. Though, this is another approximation, we can ensure not to worsen the quality by taking nodes that are a certain distance apart. Finally, we also parallelized the implementation, which accelerated the calculation notably because our testing machine contains 24 cores and many parts of the computation are easily run in parallel.

To get an overview of the performance of the implementation, we conducted different tests and also evaluated the effects the different improvements have. The results show that all the improvements are useful and all contribute greatly to the practicability of the implementation. Furthermore, we ensured the good quality of the results regarding several aspects. Even though the theoretical analysis suggests that bad results are possible, these cases do not arise in practice. Quite contrary to the theoretical results, the practical results hint at a constant upper bound of the size of the hitting set, in regard to the optimal solution. The computation of a positioning on the graph of Germany is unproblematic, taking less than half of the memory of our testing machine and runs in less than one day, still maintaining an excellent instance-based bound.

Future work

As the field of EV routing is rather unexplored, there are a lot of possible future works. The most obvious one is the improvement of the procedure that was presented in this thesis. Especially when it comes to computing the set system, there is much potential for progress. In our implementation we always constructed the paths to be hit by first exploring the unmodified graph. Algorithms for computing the CH sets directly are possible but not trivial, because the definition of ev-feasibility can not be naively applied to CH paths. The *single-source shortest path problem* for CH paths was studied in [DGNW12] with excellent results and their approach might also be applicable to our setting.

By design of our problem we chose to only hit long distance paths. This approach ignores short distance commuters and other usage of EVs where no long distance routes are used regularly. Though this problem seems to be rather a part of traffic engineering, it might also be solved in a more algorithmic way, taking several properties of the given graph into account. Another aspect of the positioning we chose willingly, is the hitting of directed paths. This approach is unrealistic when it comes to a road with two traffic lanes that are not separated physically,

but are separately represented in the graph data. It would be reasonable to only place one BLS on one side of the road. There are different other examples where this behavior would be preferred.

Instead of only solving this problem we can also solve a related one, which might implicitly take care of the former. By thickening the paths, meaning that we also allow little detours from the shortest paths to recharge the battery, the size of the set of the BLSs can be made smaller, preventing BLSs from being placed right next to each other due to special structures in the graph. Note that this is also a useful approach with regard to the practical application, as drivers generally accept little detours to refuel or recharge their vehicles. A critical point in this approach seems to be the compressed representation of the paths, because a naive CH representation is not possible when the paths are not necessarily shortest paths. Moving away from the storage of all minimal ev-infeasible paths, to an implicit hitting of detours during the hitting set algorithm, might be a way to solve this problem. Still, a notable amount of work should be necessary.

APPENDIX A

Appendix

Deutsche Zusammenfassung

Da Elektroautos immer mehr an Popularität gewinnen, ist auch ein dementsprechend ausgebautes Netz von Elektrotankstellen nötig, um dem Bedarf gerecht zu werden. Hauptsächlich längere Routen haben noch immer keine ausreichende Abdeckung, was teilweise aus der noch relativ geringen Reichweite von Elektroautos resultiert. In dieser Bachelorarbeit entwickle ich einen Algorithmus zur Platzierung von Elektrotankstellen, der das Fahren jedes beliebigen kürzesten Weges ermöglicht, in Bezug auf eine gegebene anfängliche und maximale Batterieladung. Dadurch, dass eine initiale Platzierung von Ladestationen angenommen wird, wird darauf abgezielt die Menge der Stationen so klein wie möglich zu halten. Um präzise Analysen und Auswertungen zu ermöglichen, wird ein entsprechendes Hitting Set Problem formuliert, welches dann als Basis für einen ersten naiven Algorithmus verwendet wird, der im Laufe der Bachelorarbeit erheblich verbessert wird. Des Weiteren verwende ich das duale Problem, um instanzbasierte untere Schranken zu berechnen. Abschließend wird die Implementierung getestet; im Praktischen auf Speicherverbrauch, Laufzeit und Qualität der Ladestellen Platzierung und im Theoretischen, indem allgemeine Schranken bewiesen werden. Der finale Algorithmus kann eine valide Menge von Ladestationen auf dem Graph von Deutschland in unter einem Tag berechnen, bei der die Qualität durch eine instanzbasierte Schranke gesichert ist.

Bibliography

[ADGW10] I. Abraham, D. Delling, A. V. Goldberg, R. F. F. Werneck. Alternative Routes in Road Networks. In *SEA*, pp. 23–34. 2010. (Cited on page 18)

[AHLS10] A. Artmeier, J. Haselmayr, M. Leucker, M. Sachenbacher. The shortest path problem revisited: optimal routing for electric vehicles. In *Proceedings of the 33rd annual German conference on Advances in artificial intelligence*, KI'10, pp. 309–316. Springer-Verlag, Berlin, Heidelberg, 2010. URL `http://dl.acm.org/citation.cfm?id=1882150.1882190`. (Cited on page 12)

[Chv79] V. Chvatal. A Greedy Heuristic for the Set-Covering Problem. *Mathematics of Operations Research*, 4(3):233–235, 1979. doi:10.2307/3689577. (Cited on pages 16 and 41)

[DGNW12] D. Delling, A. V. Goldberg, A. Nowatzyk, R. F. Werneck. PHAST: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing*, (0):–, 2012. doi:10.1016/j.jpdc.2012.02.007. URL `http://www.sciencedirect.com/science/article/pii/S074373151200041X`. (Cited on pages 18, 32 and 48)

[DIJ59] E. DIJKSTRA. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959. URL `http://eudml.org/doc/131436`. (Cited on pages 11 and 12)

[EF12] J. Eisner, S. Funke. Transit Nodes - Lower Bounds and Refined Construction. In *14th Meeting on Algorithm Engineering and Experiments (ALENEX)*, pp. 141–149. 2012. (Cited on pages 16 and 17)

[EFS11] J. Eisner, S. Funke, S. Storandt. Optimal Route Planning for Electric Vehicles in Large Networks. In *25th Conf. on Artificial Intelligence (AAAI)*. 2011. (Cited on page 12)

[GSSD08] R. Geisberger, P. Sanders, D. Schultes, D. Delling. Contraction hierarchies: faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th international conference on Experimental algorithms*, WEA'08, pp. 319–333. Springer-Verlag, Berlin, Heidelberg, 2008. URL http://dl.acm.org/citation.cfm?id=1788888.1788912. (Cited on page 18)

[Kar72] R. M. Karp. Reducibility Among Combinatorial Problems. In *Complexity of Computer Computations*, pp. 85–103. 1972. (Cited on page 16)

[KLS+09] M. Kuby, L. Lines, R. Schultz, Z. Xie, J. Kim, S. Lim. Optimization of hydrogen stations in Florida using the Flow-Refueling Location Model. *International journal of hydrogen energy*, 34(15):6045–6064, 2009. (Cited on page 12)

[Mina] Mineralölwirtschaftsverband e. V. Entwicklung des Tankstellenbestandes. URL http://www.mwv.de/index.php/daten/statistikenpreise/?loc=14. (Cited on page 45)

[Minb] Mineralölwirtschaftsverband e. V. Straßentankstellen in Deutschland nach Gesellschaften. URL http://www.mwv.de/index.php/daten/statistikenpreise/?loc=15. (Cited on page 45)

[MRMZ11] C. McPherson, J. Richardson, O. McLennan, G. Zippel. Planning an Electric Vehicle Battery-Switch Network for Australia. In *Australasian Transport Research Forum 2011 Proceedings*. 2011. (Cited on page 12)

[SF12] S. Storandt, S. Funke. Cruising with a Battery-Powered Vehicle and not Getting Stranded. In *26th Conf. on Artificial Intelligence (AAAI)*. 2012. (Cited on pages 12 and 15)

All links were last followed on May 22, 2013.

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

(André Nusser)