Karlsruhe Institute of Technology

Electric Vehicle Routing with Realistic Recharging Models

Master Thesis of

Tobias Zündorf

At the Department of Informatics
Institute of Theoretical Computer Science

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. Peter Sanders
Advisors:	Moritz Baum, M.Sc.
	Dipl.-Inform. Julian Dibbelt
	Dr. Andreas Gemsa

Time Period: 1st June 2014 – 24th November 2014

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Karlsruhe, 24th November 2014

Abstract

We study the problem of electric vehicle routing in road networks. Careful route planning is important, especially when planning far trips, because the battery capacity of electric vehicles is limited and charging stations are rare compared to gas stations. This makes it necessary to plan the route and all charging stops beforehand, so that the battery does not deplete along the way. While considering the battery's state of charge is important, it is still desirable to reach the target as fast as possible. In this thesis we address the problem of finding a route with minimal travel time which never violates battery constraints. We extend previous work on this problem by realistically modeled charging stations. Thereby, we take different types of charging stations into account. Our approach is capable of modeling battery swapping stations as well as regular charging stations with various charging power supplies. Our algorithm is based on a multi-objective search using Pareto-sets. Since the size of these Pareto-sets can get very large in practice, we propose several speed-up techniques in order to make the algorithm practicable. Furthermore we show heuristics leading to even faster algorithms. This thesis is concluded with a detailed evaluation of the proposed algorithm. Our experiments show that our algorithms can compete with existing algorithms, which can only solve certain special cases of our problem. Furthermore using our heuristics it is for the first time possible to find a fast and feasible routes on a continental graph.

Deutsche Zusammenfassung

Wir untersuchen das Problem der Routenplanung für Elektroautos in Straßennetzwerken. Das sorgsame Planen der Route ist im Besonderen auf längeren Strecken wichtig, da die Akkukapazität begrenzt ist und Ladestationen im Vergleich zu herkömmlichen Tankstellen selten sind. Daher ist es wichtig, die Route mit allen Ladestopps im Vorfeld zu planen, um zu verhindern, dass die Batterie unterwegs erschöpft wird. Während es wichtig ist, den Ladezustand des Akkus zu berücksichtigen, gilt es weiterhin, das Ziel so schnell wie möglich zu erreichen. In dieser Arbeit befassen wir uns mit dem Problem eine Route mit minimaler Reisezeit zu finden, welche nie die Akku-Randbedingungen verletzt. Wir erweitern frühere Arbeiten zu diesem Problem um realistisch modellierte Ladestationen. Dabei berücksichtigen wir verschiedene Arten von Ladestationen. Unser Ansatz ermöglicht die Modellierung von Battery swapping sowie normale Ladestationen mit verschiedenen Ladeleistungen. Unser Algorithmus basiert auf einer Suche mit mehreren Zielfunktionen, welche Pareto Mengen benutzt. Da diese Pareto Mengen in der Praxis sehr groß werden können, schlagen wir verschiedene Beschleunigungstechniken vor, welche den Algorithmus praktikabel machen. Darüber hinaus stellen wir Heuristiken vor, die zu noch schnelleren Algorithmen führen. Wir schließen diese Arbeit mit einer detaillierten Auswertung der vorgestellten Algorithmen ab. Unsere Experimente zeigen das unser Algorithmus mit existierenden Algorithmen, welche nur Spezialfälle von unserem Problem lösen, konkurrieren kann. Darüber hinaus ist es durch die Nutzung unserer Heuristiken zum ersten Mal mögliche schnelle und zulässige Routen für kontinentale Graphen zu berechnen.

Contents

1. Introduction

The importance of electromobility has increased steadily over the past few years. The development of electric vehicles (EV) made great progress and many EVs are already in use. While they have undeniable advantages over conventional vehicles, such as their independence of fossil fuels, there is still the huge disadvantage of their limited battery capacity. This gives rise to the demand of new route planning algorithms, which take into account the special properties of EVs.

Route planning for EVs differs in many points from conventional route planning. First of all, there is a limited capacity of the vehicle's battery, which leads to limited driving ranges of about 100 km for most EVs. Furthermore charging stations (CS) for electric vehicles are not as common as gas stations, and when used the recharging process can take several hours. Because of this it can be reasonable to use a slower route to reach the target if this saves some energy. This means that it is crucial to keep track of the state of charge (SoC) of the vehicle's battery, when searching for a fast route. The problem gets even more complicated since the SoC does not necessarily decrease when driving. The reason for this is the EVs capability of recuperating energy when driving downhill, unless the battery's capacity is exceeded.

There has already been some research on route planning for EVs, which took into account the constraints caused by limited battery capacity. We will present these research in section 1.1 in greater detail. The previous research focused primarily on finding most energy efficient routes or fastest routes which satisfy the battery constraints. However, hardly any research engaged the possibility of using charging stations for recharging the battery. A reason for this was, that recharging takes too long and is therefore not viable en route. Beyond that, recharging has only be considered as a static event which restores the complete battery in a short and constant time [Sto12a, GP14, SBW12]. This is reasonable if battery swapping stations (BSS) are available, where the complete battery gets exchanged with a new fully charged battery. But up to now these stations are only a concept.

For long distance routes it is unavoidable to use regular charging stations. This establishes the need for route planning algorithms which can handle currently available charging stations.

1.1 Related Work

A great amount of research has already addressed the field of route planning. A survey on recent route planning algorithms is given by Bast et al. [BDG$^+$14]. Many of the shown algorithms are based on Dijkstra's algorithm [Dij59], which solves the most basic routing problem, i.e., finding the shortest route between two vertices. The algorithm finds the optimal route under a single and static metric (e.g. travel time). The survey covers many well studied speedup techniques which are capable of accelerating Dijkstra's Algorithm significantly. The main principle of these techniques is to compute additional information in a preprocessing step, which then is used to accelerate the query algorithm.

Speedup Techniques. The surveyed speedup techniques are classified as either goal directed approaches or hierarchical ones.

Goal-directed techniques try to guide the search in the direction of the target. This can, for example, be done by labeling edges with additional information which specifies if an edge might be useful in order to reach a specific destination region. This approach is known as the Arc-Flags algorithm [KMS06]. Another example for a goal directed technique is A* Search [HNR68]. This technique uses a function which gives a lower bound for the remaining distance to the target. This lower bound is then used to favor roads which lead in the right direction i.e., the value of the lower bound decreases. One possibility for obtaining such a lower bound is the ALT (A*, Landmarks, Triangle inequality) algorithm [GH05]. This algorithm uses a set of landmarks and precomputes the distance between them and all other points. The query algorithm then uses these landmarks and the triangle inequality in order to compute a lower bound for the distance to the target.

Hierarchical techniques often make use of the road network's structure. If the target is far away, the fastest route uses almost always highways. The Transit Node Routing (TNR) algorithm is based on this observation [BFSS07]. It first computes a small set of transit nodes (e.g. motorway accesses) and all pairwise distances between them. Afterwards every node in the network is associated with a set of access nodes. Access nodes are those transit nodes which are relevant for the associated node. If source and target are far apart from each other, then the query algorithm needs to consider only their access nodes and the distance can be minimized using only a few table lookups. Another important approach are Contraction Hierarchies (CH) [GSSD08]. The basic idea of this technique is to remove unimportant vertices successively without changing the minimal distances between all other vertices. This is achieved by inserting new edges if the distance between two other vertices would otherwise increase. The query algorithm starts from source and target simultaneously and proceeds only from less important to more important vertices. Geisberger et al. showed that this always yields the shortest path.

In many cases it is possible to combine two or more of the basic speedup techniques. An example for this is Core-ALT [BDS$^+$08], which combines the hierarchical approach CH with the the goal directed technique ALT. The preprocessing for this algorithm contracts, just as the CH preprocessing, unimportant parts of the road network. But instead of contracting the complete network a small core remains uncontracted (e.g. 1% of the road network). Afterwards ALT is performed on the core network.

The survey covers also many other variations of the route planning problem. Some of them involve time-dependent metrics or are able to handle dynamic scenarios such as the current traffic situation. Another variation of the problem concerns the case where more than one metric has to be optimized. A common approach to handle multiple objectives utilizes Pareto sets [TC92, Mar84, SM13]. While this problem is hard in theory [GJ90] it turns out to be often feasible in practice for some transportation networks [MHW01].

EV Routing. When considering EV routing, most works consider two criteria associated with every road. One is the travel or driving time, the other is the energy consumption. Most research on EV routing focuses on integrating battery constraints into classical single criterion routing algorithms. The constraints arise from the fact that the battery capacity is limited and it is impossible to use further energy when the battery is depleted (under-charging). On the other hand EVs are capable of recuperating energy when driving downhill. This energy can be used to replenish the battery as long as its capacity limit is not reached (over-charging).

Integrating these battery constraints already leads to several distinguishable problems. One of them is the optimization of energy consumption [EFS11, SLAH11, BDPW13]. However, solutions for this problem might be undesirable in practice, since they can feature long detours in order to save energy. One solution for this problem is, to allow only such routes, where the driving time does not exceed the minimal unconstrained driving time, by more than a predetermined factor [Sto12b]. Another problem variation optimizes driving time while only such routes are allowed were the battery constraints are met [Sto12b]. The last two problems are extensions of the constrained shortest path problem (CSP) which is NP-complete [HZ80].

It is possible to adapt common speedup techniques like A* search or CH for CSP [Sto12c]. These adaptions are often also applicable for EV routing problems. It has been shown that battery constraints can be modeled by replacing the constant energy consumption per road with an energy consumption function which maps the current state of charge onto actual consumption with respect to over- and under-charging [BDPW13]. Doing so enables also the CH preprocessing technique to be used for EV routing problems [Sto12b].

All variations of the route planning and EV routing problem assume a fixed driving speed per road. Considering the EV routing problem, where the fastest route which respects the battery constraints is searched for, this might lead to inappropriate results. The computed route might utilize many small roads in order to save energy. This is not pleasing if the same result can be achieved by using larger streets but driving below the speed limit. This problem can be solved by labeling each road with multiple driving times and associated energy consumption [BDHS$^+$14].

Charging Stations. Another meaningful extension of routing problems for EVs, is the introduction of charging stations. All previous problem variations had a restricted cruising range imposed by the limited battery capacity, leading to parts of the road network which are simply not reachable by the EV. However, it is of course possible to recharge the battery by using a charging station in the same way a conventional vehicle can refuel using a gas station. While the recharging process can take a long time for EVs, the possibility should certainly be considered.

Some research has already addressed this problem. First of all, there is an extension of CSP which allows the constrained resource to be replenished without referring to the special case of EV routing [SBW12]. Furthermore there has been some research addressing the specific problem of incorporating charging stations for EVs into routing algorithms [Sto12b, GP14]. All of these approaches have in common that they present practical algorithms at the drawback that they simplify the recharging process. It is assumed that recharging takes a constant time and always results in a fully charged battery.

While this might be reasonable for battery swapping stations, it is insufficient for modeling currently available charging stations. When using regular charging stations, it is possible to interrupt the charging process at any given time. Furthermore the charging rate depends on the current SoC and decreases as the SoC approaches the battery limit. Because of this, leaving a charging station before the battery is fully charged, can lead to faster routes.

The approach of Liu et al. [LWL14] models recharging more detailed and allows to interrupt the charging process as soon as enough energy has been charged. They use a linear function to model the charging process between 0% and 80% SoC. They ignore the possibility of charging more than 80% because at this point the charging process loses efficiency significantly. While their model is reasonable, their experimental section suggests that the algorithm is only applicable for very small graphs. This renders the algorithm unusable for real world applications.

Another approach is shown by Sweda et al. [SDK14]. They also use a linear function to model the first part of the charging process. Furthermore they allow to charge the battery up to 100% and use an exponential function to model the decline of the charging rate as the SoC approaches its limit. However, their algorithm is restricted to work only on grid graphs. Their algorithm is similar to an exhaustive search and they do not report the algorithm's performance. Therefore it is not possible to adopt their solution in order to obtain an efficient algorithm for large real-world networks.

1.2 Contribution

In this thesis we address the problem of finding a path which minimizes an electric vehicle's travel time between two points in a road network. Our approach considers battery constraints as well as the opportunity of recharging the battery at charging stations.

We especially focus on modeling the charging process. Our solutions use piecewise linear convex functions in order to model the SoC-gain over time. This type of functions allows the development of efficient algorithms while they are capable of representing many different types of charging stations accurately. This includes battery swapping stations, regular charging stations with various charging powers, as well as superchargers where the maximal achievable SoC is limited due to technical restrictions.

We first develop a baseline algorithm and show how piecewise linear charging functions can be incorporated into a shortest path search. Afterwards we adapt known speedup techniques to our scenario. This results in practical algorithms, so that it is possible for the first time to solve the problem even for road networks of continental scale.

In order to obtain even faster algorithms we propose several heuristic approaches. While non of these heuristics can guarantee to find the optimal solution, we show that some of them often find the optimum. As we introduce heuristics with greater inaccuracy we observe decreasing (empirical) running times.

Finally, we evaluate the performance of all our algorithms on large real world networks. Furthermore, we analyze the accuracy of our heuristic solutions by comparing them with the exact ones.

1.3 Outline

The remainder of this thesis is structured as shown in the following overview.

Chapter 2. The preliminaries chapter lays the foundation for this work and establishes the notation used throughout this thesis. Furthermore, we introduce algorithms solving the shortest path problem and the constrained shortest path problem. We analyze their functionality in detail as it serves as a basis for our own algorithms. This chapter also covers the well-known speedup techniques A* and CH in detail, since we adapt them later for our EV routing problem.

Chapter 3. The problem statement chapter gives a precise definition of the routing problem covered in this thesis. Furthermore, we introduce charging functions and show how they can be used in order too model several types of charging stations.

Chapter 4. This chapter states our basic approach. We introduce a first algorithm capable of solving the problem described in chapter 3. We use Dijkstra's algorithm as a starting point and show step-by-step how different parts of the algorithm have to be adapted in order to incorporate charging functions.

Chapter 5. In this chapter we examine several advanced speedup techniques in order to make our algorithm applicable for larger real world networks. We start by applying an A* search. Here we show how the special properties of our problems can be exploited in order to compute excellent potential functions for the A* search. Furthermore, we describe how Contraction Hierarchies can be used in our context. Finally, we combine both speedup techniques and obtain our best correct algorithm.

Chapter 6. In this chapter we introduce some heuristic approaches. Algorithms proposed in this chapter do not guarantee to find the optimal solution. Instead, we aim for fast algorithms which only have to find a feasible solution. We present a new and intuitive approach as well as extensions to our algorithm from the previous chapters.

Chapter 7. We perform an extensive evaluation of all presented algorithms in the Experiments chapter. We start by describing the experimental setup, which includes the input data used in the experiments. Afterwards, we analyze the performance of our algorithms.

Chapter 8. This thesis is concluded with a summary of the achieved results. Furthermore, we give an outlook on future work in the field of fast electric vehicle routing.

2. Preliminaries

In this chapter we define some basic concepts and notions, which are the foundation of the following work. Furthermore, we introduce some algorithms which are adapted in subsequent chapters.

2.1 Graph Theory

Graphs. A *Graph* $G = (V, E)$ is a tuple consisting of a finite set V of *vertices* and a finite set $E \subseteq V \times V$ of *edges*. A tuple *edge* $= (u, v) \in E$ is called an directed edge from u to v. In this case v is called the *head* of the edge e and u is called its *tail*. Throughout this thesis we only consider *directed* graphs, i.e., the existence of an edge from u to v does not imply that there is an edge from v to u. A graph $G' = (V', E')$ is called a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$ holds.

The number of vertices contained in the graph is denoted by $n := |V|$, the number of edges is denoted by $m := |E|$. Given a vertex $u \in V$, the number of edges with u as tail is called the *out-degree* of u, it is formally defined as $\deg_{\text{out}}(u) := |\{(u, v) \in E\}|$. Accordingly, the number of edges with u as head, the *in-degree* of u, is accordingly defined as $\deg_{\text{in}}(u) := |\{(v, u) \in E\}|$. The total number of edges incident to u is called *degree* of u, it is defined as $\deg(u) := \deg_{\text{out}}(u) + \deg_{\text{in}}(u)$.

A scalar *edge weight function* is a function which assigns a value to each edge in a graph. In the context of electric vehicle routing we will mainly use two edge weight functions. The *driving time* $\text{dt}\colon E \to \mathbb{R}_{\geq 0}$ assigns to each edge the non-negative time it takes to traverse the edge using the electric vehicle. The energy consumption $\text{cons}\colon E \to \mathbb{R}$ assigns to each edge the energy consumed when traversing the edge. Note that the energy consumption can be negative due to recuperation.

Paths. A *path* P in G is a sequence of vertices $(u_1, u_2, \ldots, u_k)$ in V such that for $1 < i \leq k$ the edge $e = (u_{i-1}, u_i)$ exists in E. Given two vertices $u, v \in V$, if there exists any path $P = (u, \ldots, v)$ starting at u and ending at v, then v is called *reachable* from u. Any path from u to v is also called a u-v-path. If there exists an u-v-path as well as a v-u-path, then we call u and v *connected*. A path with $u_1 = u_k$ is also called a *cycle*. The number of vertices contained in a path P is denoted by $|P| = k$.

A path $Q = (v_1, v_2, \ldots, v_\ell)$ is called a *subpath* of P (denoted $Q \subset P$) if and only if $\ell \leq k$ and there exists an $i \in [0, k - \ell]$ such that for all $0 < j \leq \ell$ the condition $u_{i+j} = v_j$ holds. The special subpath of P containing the first i vertices of P is denoted by $P^i := (u_1, u_2, \ldots, u_i)$ is called a *prefix* of P.

Given an edge weight function ω on E it is possible to extend the definition of ω to paths in G. The weight of a path P regarding ω is then defined as $\omega(P) = \sum_{i=2}^{k} \omega((v_{i-1}, v_i))$.

A *tree* $\mathcal{T}_u$ *rooted* at u is a special graph with $m = n - 1$ edges such that every vertex v in $\mathcal{T}_u$ is reachable from u. This is equivalent to the fact that there exists a unique path from u to every other vertex v. The root vertex u is the only vertex in $\mathcal{T}_u$ with in-degree zero. If a vertex v in $\mathcal{T}_u$ has no outgoing edges i.e., $\deg_{\text{out}}(v) = 0$, it is called a *leaf.*

Shortest Paths. Given two vertices $u, v \in V$ and an edge weight ω (in this context often referred to as *length*) the distance between u and v is the minimal weight or length of any path starting at vertex u and ending at v. Formally the distance is defined as $\text{dist}_\omega(u, v) := \inf_{P=(u,\ldots,v)} \omega(P)$. The condition $\text{dist}_\omega(u, v) \in \mathbb{R}$ does not hold in general. If u is not connected to v, the set of paths starting at u and ending at v is empty, therefore $\text{dist}_\omega(u, v)$ is ∞. Furthermore $\text{dist}_\omega(u, v)$ may be $-\infty$ if the graph contains a cycle of negative length. In the case of $\text{dist}_\omega(u, v)$ being a real number, there exists at least one path in G starting at u and ending at v of length $\text{dist}_\omega(u, v)$. Every such path is called a *shortest path* from u to v ($\text{SP}_\omega(u, v)$).

A *shortest path tree* $\mathcal{T}_v$ rooted at v is a special subgraph of G containing all vertices reachable from v. Furthermore, $\mathcal{T}_v$ contains exactly one path from v to every other vertex and this path is a shortest path in G. The in-degree of every vertex contained in $\mathcal{T}_v$ is 1 except for u, which has an in-degree of 0. Note that $\mathcal{T}_v$ is a tree by construction.

2.2 The Shortest Path Problem

In this section, we introduce several problems related to shortest paths and algorithms capable of solving them.

There are three basic problems associated with the computation of shortest paths. These problems are listed below.

- The Single-Pair Shortest Path (SPSP) problem. We are given a graph $G = (V, E)$, an edge weight ω as well as source and target vertices $s, t \in V$. The problem asks for the distance $\text{dist}_\omega(s, t)$.
- The Single-Source Shortest Path (SSSP) problem. We are given a graph $G = (V, E)$, an edge weight ω as well as a source $s \in V$. We ask for the distance from s to every other vertex in V.
- The All-Pairs Shortest Path (APSP) problem. We are given a graph $G = (V, E)$ and an edge weight ω. The problem asks for the distance between each pair of vertices from V.

In addition to the distance it is often also asked for an example of a shortest path. This thesis focuses on extensions of the Single-Pair Shortest Path problem, but we will also encounter SSSP problems in order to solve our main problem. Both problems, SPSP and SSSP, are solved by Dijkstra's algorithm, therefore we discuss it next.

2.2.1 Dijkstra's Algorithm

A well known algorithm for solving the single-pair shortest path problem as well as the single-source shortest path problem, under the precondition that $\omega(e)$ is not negative for all $e \in E$, is Dijkstra's algorithm. The algorithm was first published by Edsger Dijkstra in 1959 [Dij59]. A pseudo code version of the algorithm is given in algorithm 2.1. Besides

Algorithm 2.1: Dijkstra's Algorithm

```
Input: Graph $G = (V, E)$, edge weight $\omega$, source vertex $s$
Data: Priority queue Q
Output: Distances from $s$ given by dist[$\cdot$], shortest-path tree given by parent[$\cdot$]

1  for each $v \in V$ do                                        // initialization
2      dist[$v$] $\leftarrow \infty$
3      parent[$v$] $\leftarrow \perp$
4  dist[$s$] $\leftarrow 0$
5  Q.insert($s$, dist[$s$])

6  while not Q.isEmpty() do                                     // main loop
7      $u \leftarrow$ Q.deleteMin()
8      for each $(u, v) \in E$ do
9          if dist[$v$] $>$ dist[$u$] $+ \omega((u, v))$ then
10             dist[$v$] $\leftarrow$ dist[$u$] $+ \omega((u, v))$
11             parent[$v$] $\leftarrow u$
12             if Q.contains($v$) then
13                 Q.decreaseKey($v$, dist[$v$])
14             else
15                 Q.insert($v$, dist[$v$])
```

calculating the distance between s and every other vertex it also outputs an implicit representation of a shortest path tree, containing a shortest path from s to every other vertex.

As input the algorithm gets a graph $G = (V, E)$, an edge weight $\omega\colon E \to \mathbb{R}_{\geq 0}$ and a source vertex s. The two arrays dist[$\cdot$] and parent[$\cdot$] are the output of the algorithm. After the algorithm terminated dist[v] holds the correct distance from s to v for every $v \in V$. The parent array implies a shortest path tree rooted at s. This means that for every vertex $v \in V$ reachable from s, which is equivalent to dist[v] $< \infty$, the shortest path from s to v can be reconstructed by following the pointers given by parent[$\cdot$] from v to s. Thus $\mathrm{SP}_\omega(s, v) = (s = v_1, v_2, \ldots, v_k = v)$ is given by $v_{i-1} :=$ parent[v_i] for all $1 < i \leq k$.

The algorithm makes use of a priority queue data structure Q. This queue maintains key-value pairs, where the value represents a vertex $v \in V$ and the key represents the tentative distance from s to v. Implementation details of the priority queue are not important for the understanding of Dijkstra's algorithm. However, the priority queue must match the following interface.

- isEmpty() returns a boolean value, which is `true` if and only if the queue holds at least one element.
- deleteMin() returns the vertex v associated with the minimal key currently contained in the queue. Afterwards v is removed from the queue.
- contains(v) returns a boolean value, which is `true` if and only if v is held by the queue.
- decreaseKey(v, d) sets the key associated with v to d. Preconditions for this operation are, that v is already contained in the queue and that the key currently associated with v is greater than or equal to d.
- insert(v, d) inserts v into the queue and sets its key to d. Precondition for this operation is, that v is not contained in the queue.

The basic approach of Dijkstra's algorithm is to grow a graph-theoretic circle around s, such that for every vertex v contained in this circle the correct distance form s to v is known and held in dist$[v]$. Furthermore parent$[\cdot]$ implies a correct shortest path tree for all vertices within this circle. The algorithm starts by initializing the arrays dist and parent, as well as the priority queue Q. Since s is the only vertex for which the distance from s is known at the beginning, dist$[s]$ is set to 0. For all other vertices it is set to dist$[v] = \infty$. Accordingly, the parent array is initialized with all entries set to $\perp$, representing an empty shortest path tree. The queue Q is initialized by inserting the source vertex s into it. After the initialization, the main loop of the algorithm begins in line 6 of algorithm 2.1. As long as there are vertices contained in Q, the vertex u with minimal key, and therefore minimal distance from s, will be removed from Q. Once a vertex has been removed from Q it is called *settled*. At this point dist$[u]$ contains the correct distance from s to u and parent$[\cdot]$ implies a shortest path from s to u. Next, the algorithm examines all outgoing edges of u. For every such edge $e = (u, v)$ it is checked if the shortest path from s to u extended by the edge e yields a shorter distance from s to v than the current tentative distance dist$[v]$. If this is the case, the edge e is *relaxed*. This means that the tentative distance of v is set to dist$[u] + \omega((u, v))$. Moreover, parent$[v]$ is set to u, since we reached v via u. Finally, v is either inserted into Q or its key is decreased, depending on it being previously contained in Q or not. Once a vertex got inserted into Q it is called *visited*.

Backwards Search. Sometimes we are not interested in the distances from one source vertex to every other vertex contained in V, but in the distances from every vertex $v \in V$ to a single given vertex t. It is possible to compute such a *backwards* shortest path tree using Dijkstra's algorithm. Similar to the forward version of the algorithm, we start by initializing all labels with distance ∞, except for the vertex t, which has its distance set to 0. Afterwards the algorithm proceeds along the lines of the forward version, except for one little change when settling a vertex u. Instead of relaxing all outgoing edges $e = (u, v)$, the backward version of the algorithm relaxes all incoming edges $e = (v, u)$. The remaining algorithm is unchanged.

Correctness. We now give a proof for the correctness of Dijkstra's algorithm i.e., we prove that, after the algorithm terminated dist$[v] = \text{dist}_\omega(s, v)$ holds for all $v \in V$. The proof is subdivided into three claims: Every reachable vertex from s gets settled. Throughout the whole computation dist$[v]$ overestimates $\text{dist}_\omega(s, v)$ i.e., dist$[v] \geq \text{dist}_\omega(s, v)$. By the time v gets settled dist$[v] = \text{dist}_\omega(s, v)$ holds.

The first claim can be proven by contradiction. Assume there is a vertex $v \in V$ reachable from s which does not get settled. Since v is reachable from s there exists a path $P = (s = v_1, v_2, \ldots, v_k = v)$. We also know that s gets settled by the algorithm in the first iteration of the main loop. Therefore, there must exist an $i \in [2, k]$ such that v_{i-1} gets settled but v_i does not. But by the time v_{i-1} gets settled the edge (v_{i-1}, v_i) gets relaxed and therefore v_i gets inserted into Q. This is a contradiction to v_i not getting settled, because the algorithm terminates only after every vertex inserted into Q got settled.

The second claim can be verified by using the fact that every change of dist$[v]$ corresponds to the length of a path P from s to v in G. By definition the length of P is at least as long as the length of a shortest path from s to v, which has length $\text{dist}_\omega(s, v)$.

The third claim can again be proven by contradiction. Assume there is a vertex v, which got settled and dist$[v] > \text{dist}_\omega(s, v)$ holds. Since v is reachable from s there exists a shortest path $P = (s = v_1, v_2, \ldots, v_k = v)$ from s to v. There have to be unsettled vertices in P, otherwise dist$[v] = \omega(P) = \text{dist}_\omega(s, v)$ would hold which is a contradiction. Let v_i be the first unsettled vertex in P. After v_{i-1} got settled the edge $e = (v_{i-1}, v_i)$ got relaxed, and therefore dist$[v_i]$, which is equal to $\text{dist}_\omega(s, v_i)$, is less than dist$[v]$. But then v_i would have been settled before v, which is a contradiction. This fact is called the *label-setting* property of Dijkstra's algorithm.

Complexity. The initialization phase of the algorithm takes $\mathcal{O}(n)$ time. Since every vertex gets settled at most once the main loop of the algorithm has at most n iterations. Therefore, the operations insert$(\cdot,\cdot)$, deleteMin() and isEmpty() get called at most n times. Furthermore, the fact that a vertex gets settled only once implies that each edge gets relaxed at most once. Therefore, the operations decreaseKey$(\cdot,\cdot)$ and contains$(\cdot)$ get called at most m times. This yields an overall running time of

$$\mathcal{O}(n + (T_{\text{insert}} + T_{\text{deleteMin}} + T_{\text{isEmpty}})n + (T_{\text{decreaseKey}} + T_{\text{contains}})m).$$

The exact running time now depends on the priority queue implementation. The best known implementation for Dijkstra's algorithm is a Fibonacci Heap which yields a worst case complexity of $\mathcal{O}(m + n \log n)$ [FT87]. In practice however, a Binary Heap, which yields a worst case complexity of $\mathcal{O}((n + m) \log n)$, is often sufficient.

The algorithm shown in algorithm 2.1 solves the SSSP problem. If we are only interested in the shortest path from s to a single vertex t, the algorithm can be terminated in line 7 as soon as u equals t. This does not affect the correctness of the algorithm, since the value of dist$[u]$ does not change after u got settled. This new stopping criterion can reduce the running of the algorithm, however it will not reduce its worst case complexity.

The proof of correctness as well as the complexity analysis make use of the fact that ω has no negative values. However, Dijkstra's algorithm still computes the correct distances for an edge weight function with negative values as long as there are no negative cycles in G. In this case it is not possible to use the stopping criterion to speedup the computation for the SPSP problem. Furthermore the label-setting property does not hold anymore, instead the algorithm becomes *label-correcting*. This also implies that the worst case complexity shown above is not valid anymore. Instead the worst case complexity for graphs with negative edge weights, but without negative cycles, is exponential in the graph size [Joh73].

Dijkstra's algorithm solves the SSSP problem efficiently. However, for the SPSP problem it is in practice often to slow for real time applications on large graphs. Because of this a lot of research has been done, trying to speedup the shortest path computation. A common approach is to use additional information at run time to speedup Dijkstra's algorithm. This additional information has of course to be calculated beforehand. Speedup techniques for Dijkstra's algorithm can be divided into two major groups. One is to make the shortest path search *goal directed*. The other is to skip *unimportant* vertices or parts of the graph. We now discuss two well studied speedup techniques.

2.2.2 A* Search

The A* search algorithm is an extension of Dijkstra's algorithm first introduced in [HNR68], which makes the search goal directed. This is done by rearranging the order in which the vertices get settled. The basic idea is to prefer vertices which are closer to t. To achieve this the algorithm makes use of a heuristic potential function $\pi_t \colon V \to \mathbb{R}$, which provides further information about the minimal distance from a vertex to t. The values for $\pi_t(\cdot)$ get computed in a preprocessing, afterwards the potential is given as additional input to the query algorithm. The function π_t is then used to change the order in which the vertices get removed from the queue. This is done by using dist$[v] + \pi_t(v)$ as key for v in the queue, in lines 13 and 15 of algorithm 2.1.

In order to maintain correctness as well as the algorithms worst case complexity, the potential π_t has to be *feasible*. This is only the case if the *reduced* edge weight, which is defined as $\bar{\omega}((u,v)) := \omega((u,v)) - \pi_t(u) + \pi_t(v)$, is greater or equal to zero for every edge $(u, v) \in E$. In this case the A* search is label setting. In fact, A* processes the vertices in the same order as Dijkstra's algorithm with $\bar{\omega}$ as edge weight function does [GH05]. If additionally $\pi_t(t) = 0$ holds we say that π_t is an *admissible* heuristic, which means that it never overestimates the distance to t.

2.2.3 Contraction Hierarchies

It is an important approach for accelerating shortest path computations to exploit the hierarchical structure of road networks. Long routes tend to consist mainly of a small subset of important roads, such as highways. This insight is used in various speedup techniques. All of them have in common that they aim to skip unimportant roads so that the search algorithm primarily scans important roads. We now describe one of these hierarchical techniques in detail.

The Contraction Hierarchies (CH) algorithm was first introduced by Geisberger et al. [GSSD08]. The main component of the algorithm is an operation called *vertex contraction*, which contracts (removes) a vertex u from the graph without changing shortest distances between the reminding vertices. In order to achieve this, a new *shortcut* edge $e = (v, w)$ is inserted for every pair v, w of neighbors from u, with the property that the only shortest path from v to w contains u. The length of this shortcut is the length of the former shortest v-w-path $P = (v, u, w)$.

Preprocessing. The preprocessing step of CH orders all vertices by importance and contracts them in this order beginning with the least important one. The result of the preprocessing is the original graph augmented with all shortcuts which where added during some vertex contraction. The order in which the vertices get contracted is crucial for the performance of the later query algorithm. Therefore, it is aimed for an order which minimizes the average search space of the later query. In order to determine a good vertex ordering various heuristics have been examined. These heuristics try to minimize the search space size by minimizing the overall number of shortcuts added during preprocessing and picking the vertices uniformly from the graph. Several measurements can be used in order to achieve a good ordering, some of them involve the vertex degree or the number of shortcuts added during the next contraction or the number of neighbor vertices which already got contracted.

When performing contracting the vertex u, the algorithm has to determine for every pair v, w of neighbor vertices from u, if a shortcut is needed or not. This is done by performing a *witness search* after u got removed from the graph. This witness search uses Dijkstra's algorithm for searching for a path from v to w which is shorter than the two edges (v, u) and (u, w). If such a path is found no shortcut is needed. However, the algorithm remains correct if a shortcut is added even if a shorter path exists. Therefore it is possible to interrupt the search, e.g. after a certain number of vertices has been settled, in order to reduce the preprocessing time.

Query Algorithm. The query algorithm runs a modified version of Dijkstra's algorithm on the graph containing both, the original edges as well as the shortcuts. This search is *bidirectional*, which means that a forwards search starting at s and a backwards search starting at t run simultaneously. In contrast to plain Dijkstra the CH query considers only edges which lead from less to more important vertices. In particular this means that the forwards search relaxes only such edges $e = (u, v)$, where u got contracted before v while the backwards search uses only the ones where v got contracted before u.

If both, the forwards and the backwards search, reach the same vertex u, then the sum of the both distance labels $\text{dist}_s[u]$ and $\text{dist}_t[u]$ is an upper bound for the distance from s to t. This is simply because both labels together correspond to a complete path from s

to t via u. In addition to that Geisberger et al. [GSSD08] showed that the most important vertex v on the shortest s-t-path is always reached by both searches and that the forwards and backwards label for the vertex v hold the correct minimal distance.

Therefore, the minimal distance from s to t can be found by minimizing $\mathrm{dist}_s[v] + \mathrm{dist}_t[v]$ over all vertices v which are reached by both searches. The algorithm keeps track of the minimal tentative distance from s to t during the search. Every time a vertex gets settled by both searches, it is checked if the tentative distance can be improved. The algorithm can stop as soon as the keys of all vertices in the queue are greater than the tentative distance.

Core ALT. The CH preprocessing technique yields an enormous speedup, but preprocessing time and the size of the shortcut graph depend on the given metric [GSSD08]. Because of this it may be unpractical to compute the complete CH. An easy solution for this problem is to interrupt the CH preprocessing at some time, leaving a small part of the graph uncontracted. Further speedup can be achieved by combining hierarchical and goal directed approaches. An example for such a technique is Core-ALT [BDS+08]. This technique applies the CH preprocessing until only a small *core* graph remains uncontracted. It then uses A* together with landmarks and the triangle inequality to accelerate the query algorithm on the core graph.

2.3 The Constrained Shortest Path Problem

We now take a look at the Constrained Shortest Path (CSP) problem, which is closely related to our problem setting. This problem is an extension of the SPSP problem where an additional edge weight function rc called *resource consumption* is given. As before, the objective of the problem is to find a path of minimal length. But now the resource consumption of the solution must not exceed a certain bound.

Definition 2.1. Constrained Shortest Path (CSP)
We are given an undirected Graph $G = (V, E)$, an edge weight ω, source and target vertices $s, t \in V$, as well as resource consumption $\mathrm{rc}\colon E \to \mathbb{R}_{\geq 0}$ *and an upper bound for the resource consumption $R \in \mathbb{R}_{\geq 0}$.*

The problem asks for the shortest path $P = (s = v_1, v_2, \ldots, v_k = t) \in G$ from s to t, which does not exceed the resource bound R i.e.,

$$\mathrm{rc}(P) = \sum_{i=2}^{k} \mathrm{rc}((v_{i-1}, v_i)) \leq R.$$

In contrast to the Single-Pair Shortest Path problem, the Constrained Shortest Path problem turns out to be NP-complete as shown by Garey and Johnson [GJ90].

The objective of the EV routing problem we study is to get as fast as possible to the target. Thus, driving time is used as the edge weight, which gets minimized. The limiting resource is the electric energy stored in the vehicles battery. In contrast to the resource consumption of CSP the energy consumption of an electric vehicle is not always positive. When driving downhill the EV's engine can possibly recuperate energy. If the battery is not fully charged the recuperated energy can be stored and used later on. So in this scenario an upper bound as used in CSP is not sufficient to completely describe the battery constraints.

The upper bound for an EV's battery capacity is denoted by M. In order to simplify the notation we use 0 as lower bound for SoC. If it is undesirable to discharge the battery completely, either because of *range anxiety* (the fear of getting stranded) or because it damages the battery, then it is simply possible to define the desired lower bound for

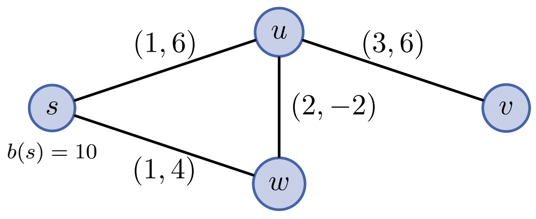

Figure 2.1: An simple example for Graph. The edges are labeled with (driving time, energy consumption). The initial SoC at s is 10, thus we want to reach u we can use the direct edge (s, u), but if we want to reach v we have to take a detour via w.

SoC as 0. Doing so has no impact on the problem's complexity and does not affect the algorithms presented in this thesis. These two bounds define the range of valid battery states $\mathrm{B} = [0, M]$.

In the context of EV routing a path is called *feasible* if the SoC of the EV's battery does not leave the range B at any given time. Checking if a given path P is feasible is slightly more complicated than checking if it obeys the resource bound of CSP. Here it was sufficient to check if $\mathrm{rc}(P) \leq R$ holds. In order to determine if P is feasible, it is not sufficient to look at the total energy consumption $\mathrm{cons}(P)$ of the path. Instead we have to look at the SoC for every vertex in P.

The initial SoC for the source vertex s is given by $b(s)$. Traversing a single edge $e = (s, v)$ reduces $b(s)$ by $\mathrm{cons}(e)$, unless this would result in overcharging the battery. In this case the SoC is set to M. So the SoC at vertex v is given by $b(v) = \max(b(s) - \mathrm{cons}(e), M)$. A negative value for $b(v)$ indicates that the initial SoC is insufficient for traversing the edge e. Using this formula we define the function $b(P, i)$ which computes the EV's SoC after traversing the first i vertices of the path $P = (v_1, v_2, \ldots, v_k)$

$$b(P,i) := \begin{cases} b(v_1) & \text{if } i = 1 \\ \min(b(P, i-1) - \mathrm{cons}((v_{i-1}, v_i)), M) & \text{else.} \end{cases}$$

The path P is called feasible if $b(P, i) \in \mathrm{B}$ holds for all $1 \leq i \leq k$. These extensions of CSP lead to the definition of the Electric Vehicle Route Planning problem.

Definition 2.2. Electric Vehicle Route Planning *(*EVR*)*
We are given a Graph $G = (V, E)$, a driving time $\mathrm{dt}\colon E \to \mathbb{R}_{\geq 0}$*, an energy consumption* $\mathrm{cons}\colon E \to \mathbb{R}$*, a range of valid battery states* $\mathrm{B} = [0, M]$*, source and target vertices $s, t \in V$, as well as the initial state of charge $b(s)$.*

The problem asks for the shortest path $P = (s = v_1, v_2, \ldots, v_k = t) \in G$ from s to t, which is feasible i.e., $b(P, i) \in \mathrm{B}$ for all $1 \leq i \leq k$.

Next we show how Dijkstra's algorithm can be modified to solve EVR. The biggest obstacle is that a subpath of a shortest path is not necessarily a shortest path on its own, when considering a constrained shortest path problem. Consider the example shown in 2.1. If the battery's capacity is 10 then the shortest feasible path from s to v has a driving time of 6. The subpath from s to u has a driving time of 3. But there exists another feasible path that reaches u with a driving time of 1. This path is perfectly fine if we want to reach u, but the remaining energy is not sufficient to reach v. So depending on which vertex is the target, we get a different driving time for reaching u. In order to solve this problem the new algorithm has to maintain more than one label for the vertex u.

2.3.1 Pareto-Sets

A common approach for solving optimization problems with multiple objectives is to utilize *Pareto*-sets, which contain all Pareto-optimal solutions [Mar84] [TC92]. A solution A is called Pareto-optimal if there exists no other solution B which is equivalent or better with respect to every objective. Otherwise we say that B *dominates* A which is denoted by $B \propto A$. In the case of EV routing our objectives are driving time and SoC. Given two paths P, Q represented by their labels $(\mathrm{dt}_P, b_P), (\mathrm{dt}_Q, b_Q)$ dominance is defined as

$$P \propto Q \quad :\Leftrightarrow \quad \mathrm{dt}_P \leq \mathrm{dt}_Q \;\wedge\; b_P \geq b_Q.$$

When solving EVR it is sufficient to consider only non-dominating paths, i.e., the Pareto-set, because a dominated path will never be part of the optimal solution. If vertex v is reachable from s via the two paths P, Q and $P \propto Q$, then Q cannot be part of the optimal path from s to t. Every v-t-path with energy consumption low enough so that it can be traversed after traversing Q can also be traversed after P, because P provides at least as much SoC at v as Q. Therefore the total s-t-path will remain feasible when using P instead of Q. Furthermore, this does not increase the driving time, since the driving time of P is at most as high as the driving time of Q.

Dijkstra's algorithm can now be changed to maintain a label-set per vertex v instead of a single label. This label-set will contain one label for every Pareto-optimal s-v-path that has been found. If an incoming edge of v gets relaxed a new label for v will be generated. This label must only be added to the label-set if it is not dominated by any previously contained label. When adding the new label, all other labels dominated by the new one, can be removed. This approach leads to the Multi-Objective Shortest Path Search as it was first introduced by Martins [Mar84].

2.3.2 Multi-Objective Shortest Path Search

The basic procedure of this algorithm is similar to Dijkstra's algorithm. As before the algorithm utilizes a queue to maintain visited but unsettled labels. These labels are settled one at a time in ascending order of their keys. Every time a label gets settled, all outgoing edges of the associated vertex are relaxed. So the basic operations of vertex settling and edge relaxations are used in the same way as they were used in Dijkstra's algorithm.

A main difference are the labels used by the algorithm. Dijkstra's algorithm used a tuple consisting of distance and parent pointer as label. Given multiple objectives the labels are changed to consist of one value for every objective and an additional parent pointer if path reconstruction is desired. Thus, for EV routing, a triple consisting of driving time, SoC and parent pointer is used as label.

Given the new labels, an order has to be defined on them. Dijkstra's algorithm settled the labels in increasing order of distance. When multiple objectives are given it is not clear how the labels should be ordered. An order can for example be defined by using a linear combination of the objectives or by using lexicographical ordering. In the special case of EVR it is the main objective to minimize the driving time, while the SoC is only used to check if the path is feasible. Therefore, the labels are ordered lexicographically with driving time as first criterion and SoC as second. Thus, labels get settled in increasing order of driving time.

Label-Sets. Dijkstra's algorithm only uses one label for each vertex v represented by $\mathrm{dist}[v]$ and $\mathrm{parent}[v]$. In contrast to that, a *label-set* $\mathrm{labelSet}[v]$ is used for every vertex v if multiple objectives are given. This label-set contains all tentative Pareto-optimal s-v-paths.

The label-set itself is organized as an array which stores all Pareto-optimal labels in the same order as they get settled.

In addition to that, each label-set maintains a pointer to the first unsettled label contained in the set. Since labels get settled in increasing order of driving time and all edges have positive driving time it is not possible that a new label is added to the label-set, which has lower driving time as an already settled label. This is analogous to the label setting property of Dijkstra's algorithm.

Algorithm 2.2 shows a pseudo code version of the multi-objective shortest path search for solving EVR. Just like Dijkstra's algorithm the algorithm starts with an initialization phase. All label-sets are initialized as empty sets, indicating that there is no path known leading to the vertex. Only the label-set for the source vertex s contains the initial label $(0, b(s), \perp)$, indicating that the search starts at s with a SoC of $b(s)$.

The label-sets used by the algorithm have to provide the three basic operations Key, Settle and HasUnsettledLabels.

- Key(labelSet) returns a key based on the first unsettled label contained in the label-set, i.e. the label with the smallest driving time of all unsettled labels contained in the label-set. As key we basically use the driving time, SoC is used in order to break ties.
- Settle(labelSet) returns the smallest (wrt. key) unsettled label contained in the label-set. Additionally this label gets marked as settled. This is done by increasing the label-set's internal pointer to the first unsettled label by one.
- HasUnsettledLabels(labelSet) returns a boolean value, which is `true` iff the label-set contains one ore more unsettled labels. This is the case if the label-set's internal pointer points to an actual label. The label-set contains no unsettled labels if the internal pointer points behind the array used to store the contained labels.

In each iteration of the main loop the vertex u with the minimal key, i.e. driving time from s to u, is removed from the queue. Afterwards the label with minimal key, of the label-set associated with u, gets settled. If the label-set of u contains further labels, the vertex u gets reinserted into the queue, using the key of the next unsettled label in the label-set. Settling a label is done by relaxing all outgoing edges of estimating Pareto-sets.the current vertex u, similar to plain Dijkstra's algorithm. When relaxing an edge $e = (u, v)$, the labels driving time and SoC get altered by the according weights of the edge e. At this point the algorithm also takes over- and under-charging into account. When computing the SoC after traversing e in line 12, the battery capacity M is used as an upper bound in order to prevent over-charging. If the SoC b' of the new label drops below 0, under-charging occurs. This means that the edge cannot be traversed, thus the new label is discarded in line 13. At the same time we check if there already exits another label which dominates the newly created one. In this case the new label is also discarded, because it can never be part of the optimal solution. If, on the other hand, the label is not dominated and has a valid SoC, then it is added to the label of vertex v and the key of v in the queue is adjusted accordingly.

Complexity. Since the the algorithm solves a variation of the constrained shortest path problem, which is NP-complete [HZ80], it is unlikely that it has a polynomial running time. Indeed it is possible that the set of Pareto optimal solution contains exponentially many labels, in the size of the graph. Since complete Pareto sets are computed during the execution of the algorithm, its running time is at least exponential. However, in practice the actual size of the involved Pareto-sets can be much smaller [MHW01], which makes the algorithm often feasible in practice.

Algorithm 2.2: MULTI-OBJECTIVE SHORTEST PATH SEARCH for EVR

```
Input: Graph $G = (V, E)$, driving time dt, energy consumption cons,
battery constraints $\mathrm{B} = [0, M]$ source and target vertices $s, t$, initial SoC $b(s)$
Data: Priority queue Q
Output: Array labelSet[·] containing a label-set for each vertex in $V$

1  for each $v \in V$ do                                          // initialization
2      labelSet[$v$] ← $\emptyset$
3  labelSet[$s$] ← $\{(0, b(s), \perp)\}$
4  Q . insert($s$, KEY(labelSet[$s$]))

5  while not Q . isEmpty() do                                     // main loop
6      $u$ ← Q . deleteMin()
7      if $u = t$ then stop
8      (dt, $b$, parent) ← SETTLE(labelSet[$u$])
9      if HASUNSETTLEDLABELS(labelSet[$u$]) then Q . insert($u$, KEY(labelSet[$u$]))
10     for each $(u, v) \in V$ do
11         dt′ ← dt + dt($u, v$)
12         $b' \leftarrow \min(b - \mathrm{cons}(u, v), M)$
13         if ($b' \in \mathrm{B}$) and not (labelSet[$v$] $\propto$ (dt′, $b'$, $u$)) then
14             labelSet[$v$] ← labelSet[$v$] $\cup$ {(dt′, $b'$, $u$)}
15             if Q . contains($v$) then
16                 Q . decreaseKey($v$, KEY(labelSet[$v$]))
17             else
18                 Q . insert($v$, KEY(labelSet[$v$]))
```

Label-Correcting Algorithm. The algorithm presented in 2.2 is label setting provided that driving time is used as primary key criterion and that the driving time for each edge is positive. In this case labels get settled in increasing order of driving time. Thus it is not possible that a label gets dominated and therefore removed, once it has been settled. But it is of course possible that multiple labels associated with the same vertex get settled during the execution of the algorithm.

The algorithm presented here settles only one label in each iteration of the main loop. The algorithm can be altered to settle the complete label-set of a vertex once the vertex gets extracted from the queue. This variation does not affect the correctness of the algorithm. But it is possible that a label that has already been settled gets dominated by another label that is added to the same label-set later. Therefore this variation of the algorithm is called label correcting.

2.3.3 Energy Consumption Functions

The multi-objective shortest path search is a basic technique for solving EVR. The Battery constraints are explicitly checked by this algorithm in lines 12 and 13. These explicit checks can get complicated when adapting advanced speedup techniques for this problem. It is, however, possible to check battery constraints implicitly by using a edge weight function.

Instead of assigning a constant energy consumption to every edge this new edge weight function assigns an *energy consumption function* to every edge. These energy consumption functions where introduced by [EFS11] and describe the effectively consumed energy depending on the SoC before traversing the edge. Formally, an energy consumption function $\mathrm{c}\colon \mathrm{B} \to \mathbb{R} \cup \{\infty\}$ is a function which maps SoC to energy consumption. The function c takes battery constraints, i.e. over- and undercharging, already into account. The special value ∞ indicates that the SoC is not sufficient for traversing the edge.

Such an energy consumption function can always be described using only three values cost, minIn and maxOut. The actual energy consumption, if neither over- nor undercharging occur, is given by cost. The value minIn defines the minimal SoC that is needed for traversing the edge. The value maxOut defines the the maximal SoC after traversing the edge, i.e., the SoC after traversing the edge with a previously fully charged battery. These three values define the energy consumption function $\mathrm{c}(b)$ (see Figure 2.2 for an example) as

$$\mathrm{c}(b) := \begin{cases} \infty & \text{if } b < \mathrm{minIn} \\ \mathrm{cost} & \text{if } b - \mathrm{cost} < \mathrm{maxOut} \\ b - \mathrm{maxOut} & \text{else.} \end{cases}$$

Energy consumption functions are used to determine the change of SoC when traversing an edge e. The SoC b' after traversing e is defined as $b' := b - \mathrm{c}_e(b)$, where b is the initial SoC and c_e is the energy consumption function for e. The definition of the energy consumption function ensures that $b' \in \mathrm{B} \cup \{\infty\}$ holds, where the special value ∞ indicates that the initial SoC was not sufficient to traverse the edge. Furthermore, the energy consumption functions fulfill the *FIFO property*, i.e., given two SoC values b_1, b_2, then $b_1, \leq b_2$ implies that $b_1 - \mathrm{c}_e(b_1) \leq b_2 - \mathrm{c}_e(b_2)$. This means that starting with a higher SoC will never result in a lower SoC after traversing an edge.

A given constant energy consumption $\mathrm{cons}(e)$ for the edge e, can by modeled with an energy consumption function by defining the three values cost_e, minIn_e and maxOut_e as

$$\begin{aligned} \mathrm{cost}_e &:= \mathrm{cons}(e) \\ \mathrm{minIn}_e &:= \max[0,\ \mathrm{cons}(e)] \\ \mathrm{maxOut}_e &:= \min[M,\ M - \mathrm{cons}(e)]. \end{aligned}$$

Energy consumption functions cannot only be used to model the energy consumption on a single edge, but also an a complete path. When considering the energy consumption of a path P, it is not enough to look at the summed energy consumption of all edges in P which is given by $\mathrm{cons}(P)$. This value may neglect effects of over- and under-charging. Consider a path P with two edges, where the first has a consumption of 1, and the second one a consumption of -1. For this example $\mathrm{cons}(Path)$ is 0 but is still not possible to to traverse the path with an empty battery, due to the first edge. In the opposite case, where the first edge has consumption -1 and the second one has consumption 1, the path is always traversable. But given the battery capacity $M = 2$, it becomes clear that the final SoC is the same for any initial SoC grater than 1. These effects can be modeled using an energy consumption function.

Linking. In order to compute the energy consumption function for a path, the *link* operation is used. Given two edges $e, f \in E$ and their energy consumption functions c_e and c_f, the linked energy consumption function $\mathrm{c}_{e\circ f}$ describes the energy consumption if the edges e and f are traversed successively. Formally, the linked energy consumption function $\mathrm{c}_{e\circ f}$ is defined as $\mathrm{c}_{e\circ f}(b) := \mathrm{c}_f(b - \mathrm{c}_e(b))$. The linked energy consumption function $\mathrm{c}_{e\circ f}$ can again be represented by the three values $\mathrm{cost}_{(e\circ f)}, \mathrm{minIn}_{(e\circ f)}$ and $\mathrm{maxOut}_{(e\circ f)}$. Given these three values for the energy consumption functions of e and f, the result of linking these edges is

$$\begin{aligned} \mathrm{minIn}_{(e\circ f)} &= \max[\mathrm{minIn}_e,\ \mathrm{minIn}_f + \mathrm{cost}_e] \\ \mathrm{maxOut}_{(e\circ f)} &= \min[\mathrm{maxOut}_f,\ \mathrm{maxOut}_e - \mathrm{cost}_f] \\ \mathrm{cost}_{(e\circ f)} &= \max[\mathrm{cost}_e + \mathrm{cost}_f,\ \mathrm{minIn}_e - \mathrm{maxOut}_f]. \end{aligned}$$

In order to be able to traverse e and f, one has to be able to traverse at least e, therefore $\mathrm{minIn}_{(e\circ f)}$ is at least as high as minIn_e. Furthermore, the remaining energy after traversing e

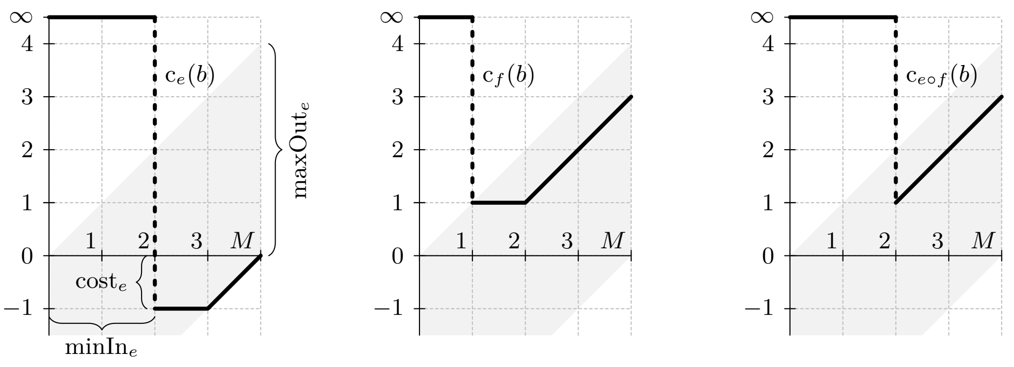

Figure 2.2: An example for two energy consumption functions c_e, c_f and the result $c_{e\circ f}$ of linking them. The battery capacity is $M := 4$. The function c_e is defined by $\text{cost}_e = -1$, $\text{minIn}_e = 2$ and $\text{maxOut}_e = 4$, the function c_f is defined by $\text{cost}_f = 1$, $\text{minIn}_f = 1$ and $\text{maxOut}_f = 1$. Linking these two functions yields $\text{cost}_{e\circ f} = 1$, $\text{minIn}_{e\circ f} = 2$ and $\text{maxOut}_{e\circ f} = 1$. Note that $\text{cost}_{e\circ f}$ is higher than the sum of cost_e and cost_f, which is 0.

has to be sufficient to traverse f. Thus $\text{minIn}_{(e\circ f)} - \text{cost}_e$ is at least as high as minIn_f, so $\text{minIn}_{(e\circ f)}$ is the maximum of this two lower bounds. Likewise, $\text{maxOut}_{(e\circ f)}$ is defined as the minimum of two upper bounds for the maximal remaining SoC. It is not possible to have a higher SoC after traversing e and f as the maximal possible SoC after traversing only f. Moreover the final SoC cannot be higher as the SoC resulting from traversing f with the maximal possible SoC after traversing e, i.e., $\text{maxOut}_{(e\circ f)} \leq \text{maxOut}_e - \text{cost}_f$.

The definition of $\text{cost}_{(e\circ f)}$ is a bit more complicated. Clearly, the sum of the consumption of both edges is a lower bound on the overall energy consumption. But in some cases the actual energy consumption will be higher than this sum, no matter what the initial SoC is. Figure 2.2 shows an example of this case. The third graph shows clearly that $\text{cost}_{e\circ f} = 1$ holds. the energy consumption function defined by $\text{cost} = \text{cost}_{edge} + \text{cost}\, f = 0$, $\text{minIn} = 2$ and $\text{maxOut} = 1$ would, however, yield the same results, since traversing both edges always involves either over- or under-charging. But if we want cost to be a tight bound for the minimal consumption of a path, we have to account for this special case.

The special case occurs when traversing the path always involves either over- or under-charging. For the energy consumption function this means that the horizontal part of the function disperses, i.e., the second case of the function's definition. The minimal energy consumption then occurs at the intersection of the two straight lines $x = \text{minIn}$ and $y = x - \text{maxOut}$, i.e., the vertical and the diagonal part of the function. Thus it becomes clear that $use_{(e\circ f)}$ cannot be less than $\text{minIn}_e - \text{maxOut}_f$.

The link operation is not commutative but associative. Therefore the energy consumption function of a path P can be computed by splitting P into arbitrary subpaths, computing the energy consumption function for these subpaths and finally link them in the same order as the subpaths occur in P.

While the summed energy consumption $\text{cons}(P)$ cannot be used to determine the actual energy needed for traversing P, it still can be useful as a lower bound for the required energy. For any path P the condition $\text{cons}(P) := \sum_{e\in P} \text{cons}(e) \leq \text{minIn}_P$ holds. Obviously this is true for a Path P containing only one edge e. In this case

$$\text{minIn}_P = \text{minIn}_e := \max[0,\ \text{cons}(e)] \geq \text{cons}(e) = \text{cons}(P)$$

holds per definition. If the path P consists of more than one edge, it can be split into the first edge e and a remaining path P'. In this case

$$\begin{aligned}\mathrm{minIn}_P &:= \max[\mathrm{minIn}_e,\ \mathrm{minIn}_{P'} + \mathrm{cost}_e] \\ &\geq \mathrm{minIn}_{P'} + \mathrm{cost}_e \\ &= \mathrm{minIn}_{P'} + \mathrm{cons}(e) \\ &\geq \mathrm{cons}_{P'} + \mathrm{cons}(e) = \mathrm{cons}(P)\end{aligned}$$

holds, thus $\mathrm{cons}(P)$ is a valid lower bound for the minimal energy minIn_P required to traverse P.

In order to adapt common single criterion speedup techniques for EV routing, energy consumption functions are often useful. We give some examples for such speedup techniques in the next section.

2.3.4 Speedup Techniques

Most of the speedup techniques presented for the shortest path problem can be adapted for EV routing or, more generally, for multi objective shortest path search. We now describe some approaches which can also be used in our scenario.

A* Search. Similar to the single criterion shortest path problem, it is possible to direct the search towards the target using A* search. Given multiple objectives, this requires multiple potential functions which provide lower bounds for each of the objectives [SI91]. When considering EV routing problems, this means that two potentials are needed. One providing a lower bound for the driving time and one providing a lower bound for the needed energy. Such lower bounds can be obtained by performing backwards Dijkstra searches on both of these objectives [HS13, SF12].

The lower bound for the energy consumption can furthermore be used to prune the search. If a label's SoC exceeds the lower bound for energy consumption, given by the associated potential, the label cannot be extended to a feasible path reaching t, thus the label can be discarded. This restricts the search space to vertices from which the target t is reachable.

Contraction Hierarchies. The CH speedup technique can also be modified to work with multiple objectives and resource constraints, as shown in [Sto12c]. While the query algorithm stays mainly unchanged, the preprocessing step and especially the vertex contraction operation become slightly more complicated. When contracting a vertex in the original CH algorithm, all shortest paths distances between its neighbors are preserved by inserting shortcuts. The length of this shortcut was determined by the sum of the length of the edges that got replaced by the shortcut.

Some changes have to be made when adapting the vertex contraction operation for multiple objectives or the EVR problem. Firstly, there exists not necessarily a unique shortest path. This means that instead of preserving shortest paths when contracting a vertex, we now have to preserve the complete Pareto set. This also means that there might be multiple Pareto optimal paths that have to be inserted between the same pair of neighbors when contracting a vertex. Therefore, the resulting shortcut graph may have multiedges. Secondly, when considering the EVR problem it is not possible to define the consumption of a shortcut as the sum of the consumption of the edges which are replaced by the shortcut, since over- and under-charging have to be considered. In order to take this into account energy consumption functions can be used to accurately represent the energy consumption of the shortcuts.

As before, when contracting a vertex u a witness search is performed for every pair v, w of neighbors of u. The shortcut from v to w can only be omitted if the graph contains a path which dominates the shortcut and does not contain u. A variation of the multi-objective shortest path search, which uses energy consumption functions, is used to search for such a path in the graph after u got removed. Unfortunately this search is significantly slower than Dijkstra's algorithm, used in the original CH preprocessing, thus the preprocessing for multiple objectives takes much longer.

It is possible to counteract this by using the fact that it is admissible for the witness search to produce false negative results. This means that the witness search determines that the shortcut is not dominated by any path in the remaining graph, although this is actually the case. Common strategies restrict the number of queue extractions, or the maximal number of labels per label-set. While this may lead to additionally and unnecessarily inserted shortcuts it can reduce the running time of the algorithm.

Combinations. Due to the fact that complete Pareto sets are assigned to every shortcut in the CH, its size increases drastically compared to a CH using only a single objective. This in turn increases the time needed to perform witness searches. Because of this it may become unpractical to compute the complete CH. A common approach to handle this problem is to interrupt the CH preprocessing at some point resulting in a partial CH and an uncontracted core graph.

In this case it is again possible to combine the CH speedup technique with another speedup technique for the core graph. An example for such a technique is the SHARC algorithm [DW09]. The algorithm combines CH vertex contractions with the Arc-Flags speedup technique for the remaining core graph, in order to solve the multi-criteria shortest path problem.

2.4 Functions

A *function* $f\colon X \to Y$ is a mapping between two sets X and Y, which assigns to each value $x \in X$ a value $f(x) \in Y$. In this context x is called the *argument* of f and $f(x)$ is called the *value* of f for x. The Set $\{(x, y) \in X \times Y \mid y = f(x)\}$ is called the *graph* of f. The graph of a function should not be confused with the graph introduced in section 2.1.

The *inverse* of a function $f\colon X \to Y$ is another function $f'\colon Y \to X$ such that the condition $f(x) = y \Leftrightarrow f'(y) = x$ is fulfilled. This means that the inverse function assigns the associated argument to each value of f and the other way around, i.e. $f'(f(x)) = x$ and $f(f'(y)) = y$. The inverse function does of course only exists if there exists for every $y \in Y$ only at most one $x \in X$ such that $f(x) = y$ holds. If this is the case, the function is called *injective.* Furthermore there must also exist at least one $x \in X$ for every y such that $f(x) = y$ holds. A function with this property is called *surjective.* Thus the inverse of a function does only exist if the function is injective and surjective.

Piecewise Linear Functions. A *piecewise linear function* $f\colon X \to Y$ is a function, such that there exists a decomposition of X into intervals with the property that f restricted to one of these intervals is a linear function. If $x \in X$ is the endpoint of one of these intervals, then we call the point $(x, f(x))$ a *supporting point* of f.

A piecewise linear function is uniquely defined by the sequence of all its supporting points, sorted by their x-values. Given such a sequence of supporting points $[(x_1, y_1), \ldots, (x_k, y_k)]$, the associated piecewise linear function f is defined as

$$f(x) := \begin{cases} \frac{(x-x_1)(y_2-y_1)}{x_2} + y_1 & \text{if } x_1 \leq x < x_2 \\ \vdots & \\ \frac{(x-x_{k-2})(y_{k-1}-y_{k-2})}{x_{k-1}} + y_{k-2} & \text{if } x_{k-2} \leq x < x_{k-1} \\ \frac{(x-x_{k-1})(y_k-y_{k-1})}{x_k} + y_{k-1} & \text{if } x_{k-1} \leq x \leq x_k. \end{cases}$$

Note that the function value of the supporting points is always defined by the linear function for the interval on their right side, except for the last supporting point. This means that, following this definition, the left endpoint of each interval is always included in the interval, while the right endpoint is only included in its interval, if it is the last one.

Slope. The *slope* of a function f at position x is defined as the value of the derivative of f at position x. The slope of a function is therefore only defined for those values of x where+f is differentiable. In this case, the slope of f at x is denoted by

$$\frac{\partial f(x')}{\partial x'}(x).$$

The slope is undefined for values of x where f is not differentiable.

Piecewise linear function are differentiable for all values of x, except for the x-values of their supporting points. This means that slope is actually not defined at the supporting points of f. Nevertheless we can assign a value to slope at these points by exploiting that the function value of a supporting point is defined by the linear function to its right. Therefore f is right differentiable for all x except the last supporting point, where it is left differentiable.

We therefore define the slope of a piecewise linear function f, which is given by its supporting points $[(x_1, y_1), \dots, (x_k, y_k)]$, at position x as

$$\frac{\partial f(x')}{\partial x'}(x),$$

where $\partial f(x')/\partial x'$ denotes the right derivative of f, if $x < x_k$ holds and the left derivation otherwise.

Convex and Concave. A function f is called *convex* if its slope is monotonically increasing, i.e., For every $x_1 \leq x_2 \in X$ the equation $\partial f(x)/\partial x(x_1) \leq \partial f(x)/\partial x(x_2)$ holds. If, on the other hand, the slope of the function f is monotonically decreasing, i.e. For every $x_1 \leq x_2 \in X$ the equation $\partial f(x)/\partial x(x_1) \geq \partial f(x)/\partial x(x_2)$ holds, then f is called *concave*. Note that a function that is not convex is not necessarily concave. If, however, f is a convex function, then $g(x) := -f(x)$ is a concave function and the other way around.

Domination. Functions can also be used in the context of Pareto optimization. If a Pareto-set previously contained points from $X \times Y$, then we know allow that it may also contain functions $f \colon X' \to Y$, where X' is a subset of X. The dominance relation for this functions is then defined as follows.

A function $f \colon X' \to Y$ dominates a single point $(x, y) \in X \times Y$ is there exists one point (x', y') in the graph of f, i.e., $f(x') = y'$, such that (x', y') dominates (x, y). This means

$$f \propto (x, y) \quad :\Leftrightarrow \quad \exists\, x' \in X' : \ (x', f(x')) \propto (x, y).$$

A point $(x, y) \in X \times Y$ dominates a function $f \colon X' \to Y$ if it dominates all the points in the graph of f, i.e.,

$$(x, y) \propto f \quad :\Leftrightarrow \quad \forall\, x' \in X' : \ (x, y) \propto (x', f(x')).$$

A function $f \colon X' \to Y$ dominates another function $g \colon X'' \to Y$ if each point in the graph of g is dominate by some point in the graph of f.

$$f \propto g \quad :\Leftrightarrow \quad \forall\, x \in X'' : \ \exists\, x' \in X' : \ (x', f(x')) \propto (x, g(x)).$$

3. Problem Statement

There has already been some research in the field of electric vehicle routing. Some algorithms exist which compute either the most energy efficient route or the fastest route regarding battery constraints. Furthermore, there has already been some research on recharging stations and how they can be handled by algorithms. One approach assumes the cost of recharging events to be constant and that it would always result in a fully charged battery afterwards. While this is adequate for battery swapping stations, it might be insufficient for regular charging stations. For instance, it is an unnecessary waste of time to recharge the battery completely, if a partially charged battery is sufficient to reach the target.

Similar to previous approaches we define *charging stations* to be a subset of the road networks vertices. We denote this set of charging stations as $\mathrm{CS} \subseteq V$. In contrast to previous approaches, we allow each charging station $v \in \mathrm{CS}$ to have its own specific charging characteristics. We do so by associating a *charging function* $\mathrm{cf}_v \colon \mathrm{B} \times \mathbb{R}_+ \to \mathrm{B}$ with every charging station v. This function takes two parameters, the initial SoC $b \in \mathrm{B}$ at which the charging station was reached, and the desired charging time $\mathrm{ct} \in \mathbb{R}_{\geq 0}$, and maps them to a resulting SoC $\mathrm{cf}_v(b, \mathrm{ct})$.

Considering the new flexibility introduced by charging functions, it becomes clear that it is no longer sufficient to determine which roads should be used and where the battery should be recharged. In order to completely describe the route, it is required to know the *charging time* spent at each station. Because of this, a solution for the problem consists not only of a path P from s to t, but also of a function $\mathrm{ct} \colon \mathrm{CS} \cap P \to \mathbb{R}_+$, which assigns a charging time to every charging station in P.

The introduction of charging stations also requires us to adapt the definition of a *feasible* path. As before, the set of valid battery states is given by $\mathrm{B} = [0, M]$, and the initial battery state at s is given by $b(s) \in \mathrm{B}$. For a given path $P = (s = v_1, v_2, \dots, v_k = t)$ and associated charging times $\mathrm{ct} \colon \mathrm{CS} \cap P \to \mathbb{R}_{\geq 0}$, we then define the EV's SoC $b(P, i)$ when arriving at vertex v_i as

$$b(P,i) := \begin{cases} b(v_0) & \text{if } i = 1 \\ \min(b(P, i-1) - \mathrm{cons}(v_{i-1}, v_i), M) & \text{if } v_{i-1} \notin \mathrm{CS} \\ \min(\mathrm{cf}_{v_{i-1}}(b(P, i-1), \mathrm{ct}(v_{i-1})) - \mathrm{cons}(v_{i-1}, v_i), M) & \text{else.} \end{cases}$$

As before this function already takes over-charging into account by using M as an upper bound for its values. A negative value for $b(P,i)$ indicates that it is not possible to reach v_i using this path. Therefore, the path P is called feasible only if $b(P,i)$ is not negative for all $1 \leq i \leq k$.

Because of the new charging stations, we also have to redefine what a shortest path is, i.e., which property of the path should be minimized. As before we want to arrive as early as possible at the target vertex. Besides driving time we now have to take into account the additional charging time. The sum of a path's driving time and charging time is called *travel time.* It is defined as

$$\mathrm{tt}(P) := \mathrm{dt}(P) + \sum_{v \in \mathrm{CS} \cap P} \mathrm{ct}(v).$$

Note that because of this $\mathrm{dt}(P) \leq \mathrm{tt}(P)$ always holds. Now that we have introduced the elements needed to describe charging stations, we can introduce the Electric Vehicle Route Planning With Recharging problem.

Definition 3.1. Electric Vehicle Route Planning With Recharging (EVRC)
We are given an directed Graph $G = (V, E)$, a driving time $\mathrm{dt}\colon E \to \mathbb{R}_+$, an energy consumption $\mathrm{cons}\colon E \to \mathbb{R}$, a range of valid battery states $\mathrm{B} = [0, M]$, a source and a target vertex $s, t \in V$, the initial battery state $b(s) \in \mathrm{B}$, as well as a set of charging stations $\mathrm{CS} \subseteq V$, and for every charging station $v \in \mathrm{CS}$ a recharging function $\mathrm{cf}_v\colon \mathrm{B} \times \mathbb{R}_+ \to \mathrm{B}$.

The problem asks for a feasible path $P = (s = v_1, v_2, \ldots, v_k = t) \in G$ which minimizes travel time, together with a function $\mathrm{ct}\colon \mathrm{CS} \cap P \to \mathbb{R}_+$ which assigns a charging time to every charging station contained in P.

The path P is feasible if and only if neither under-charging nor over-charging of the battery occur on P, i.e., $b(P,i) \in \mathrm{B}$ holds for every $1 \leq i \leq k$.

The Electric Vehicle Route Planning With Recharging problem extends the Electric Vehicle Route Planning problem, by adding charging stations. If CS is the empty set we get an instance of the original EVR problem. Therefore, every instance of CSP is also an instance of EVRC. Consequently, EVRC is also NP-hard.

3.1 Charging Functions

A charging function $\mathrm{cf}\colon \mathrm{B} \times \mathbb{R}_+ \to \mathrm{B}$ is a function which maps the current SoC $b \in \mathrm{B}$ and a desired charging time $\mathrm{ct} \in \mathbb{R}_{\geq 0}$ to the resulting SoC after the recharging process has finished.

We demand a charging function to fulfill certain conditions in order to be meaningful. First of all, a charging function should only increase the battery's SoC. This means that for an arbitrary initial SoC b and any charging time $\mathrm{ct} \geq 0$ the condition $\mathrm{cf}(b, \mathrm{ct}) \geq b$ holds. Moreover, we demand that a charging function is monotonously increasing with respect to charging time

$$\forall\, b \in \mathrm{B}: \ \mathrm{ct}_1 \leq \mathrm{ct}_2 \ \Rightarrow \ \mathrm{cf}(b, \mathrm{ct}_1) \leq \mathrm{cf}(b, \mathrm{ct}_2).$$

This means that a longer charging time never results in a less charged battery. We call a charging function which fulfills this property *monotonous.*

In reality the charging speed depends on the used amperage, which is reduced if the battery is nearly fully charged, in order to prevent the battery from getting damaged [JW06]. Because of this, the charging speed drops as the SoC approaches the battery's capacity. We presume that the decline of charging speed is again a monotonous process, i.e., a battery with a higher SoC can never be charged faster than a battery with less SoC. We call a charging function with this property *concave*, which is formally defined as

$$\forall\, b \in \mathrm{B}:\ \mathrm{ct}_1 \leq \mathrm{ct}_2 \ \Rightarrow\ \frac{\partial \mathrm{cf}(b,t)}{\partial t}(\mathrm{ct}_1) \geq \frac{\partial \mathrm{cf}(b,t)}{\partial t}(\mathrm{ct}_2).$$

If a charging function has all these three properties (i.e., it never reduces the SoC, it is monotonous and it is concave), it is called *feasible.*

Simplification. The restrictions we have introduced so far ensure that we have to consider only meaningful charging functions when designing our algorithms. Next, we would like to simplify the charging functions so that we do not need to handle two dimensional functions. Therefore, we introduce the assumption that, in order to describe a complete charging function, it is sufficient to know the charging function for an initially empty battery. All other values can then be reconstructed by shifting this function.

For this purpose we introduce a new special charging function $\mathrm{cf}'\colon \mathbb{R}_{\geq 0} \to \mathrm{B}$, which maps charging time to resulting SoC for the special case that the battery is completely depleted before the recharging process starts. Formally, this means $\mathrm{cf}'(\mathrm{ct}) = \mathrm{cf}(0, \mathrm{ct})$. We now use the function $\mathrm{cf}'(\mathrm{ct})$ to construct $\mathrm{cf}(b, \mathrm{ct})$ for any initial SoC b. We do so by adding the time, that would have been needed to charge from 0 to b, to the charging time ct. This leads to the following definition for charging functions $\mathrm{cf}(b, \mathrm{ct})$.

Definition 3.2. Charging Functions
Given a monotonously increasing function $\mathrm{cf}'\colon \mathbb{R}_{\geq 0} \to \mathrm{B}$*, which maps charging time to resulting SoC for an initially empty battery, we define the complete charging function* $\mathrm{cf}\colon \mathrm{B} \times \mathbb{R}_+ \to \mathrm{B}$ *for an arbitrary initial SoC* b *as*

$$\mathrm{cf}(b, \mathrm{ct}) := \mathrm{cf}'(\mathrm{ct} + \mathrm{cf}'^{-1}(b)).$$

In this definition we use $\mathrm{cf}'^{-1}(b)$, which is the inverse of the function $\mathrm{cf}'(\mathrm{ct})$. But, in general, a monotonously increasing function $\mathrm{cf}'(b)$ is neither injective nor surjective. Thus its inverse might not be defined.

But in Definition 3.2 we do not need a proper inverse function. We only need a function, which yields the time needed to charge from 0 to b, so that we can add this time to the charging time ct. This means we only need a function $\mathrm{cf}'(b)$, which fulfills $\mathrm{cf}'(\mathrm{cf}'^{-1}(b)) = b$, if there exists any $\mathrm{ct} \geq 0$, such that $\mathrm{cf}'(\mathrm{ct}) = b$ holds.

Therefore we define $\mathrm{cf}'^{-1}(b)$ in this case as the function, which maps SoC b onto the minimal charging time which is required to reach a SoC of at least b

$$\mathrm{cf}'^{-1}(b) := \begin{cases} \min\{\mathrm{ct} \mid \mathrm{cf}'(\mathrm{ct}) \geq b\} & \text{if } \exists\, \mathrm{ct} : \mathrm{cf}'(\mathrm{ct}) = b \\ \infty & \text{else.} \end{cases}$$

While this definition of $\mathrm{cf}(b, \mathrm{ct})$ allows us to work with simpler, one dimensional functions, it highly restricts the set of possible charging functions. However the one dimensional function cf' is sufficient to model realistic charging functions. We will see this in Section 7, when we use our functions to model charging functions originating from realistic data.

In addition to a function which maps initial SoC and charging time onto resulting SoC, we will sometimes need a function which maps initial SoC b_i and desired SoC b_d onto the minimal charging time required to surpass b_d. Using the definition above we can define the function $\mathrm{cf}^{-1}\colon \mathrm{B} \times \mathrm{B} \to \mathbb{R}_{\geq 0}$ as

$$\mathrm{cf}^{-1}(b_i, b_d) := \mathrm{cf}'^{-1}(b_d) - \mathrm{cf}'^{-1}(b_i).$$

This function will always yield the minimal charging time needed to reach at least a SoC of b_d, provided that b_d is reachable. We prove that the charging time given by $\mathrm{cf}^{-1}(b_i, b_d)$ is always sufficient to reach a SoC of b_d, by using it as argument for $\mathrm{cf}(b_i, \mathrm{ct})$.

$$\begin{aligned}
\mathrm{cf}(b_i, \mathrm{cf}^{-1}(b_i, b_d)) &= \mathrm{cf}'(\mathrm{cf}^{-1}(b_i, b_d) + \mathrm{cf}'^{-1}(b_i)) \\
&= \mathrm{cf}'(\mathrm{cf}'^{-1}(b_d) - \mathrm{cf}'^{-1}(b_i) + \mathrm{cf}'^{-1}(b_i)) \\
&= \mathrm{cf}'(\mathrm{cf}'^{-1}(b_d)) \;\geq\; b_d
\end{aligned}$$

Furthermore, we can use a similar argument to show that any charging time shorter than $\mathrm{cf}^{-1}(b_i, b_d)$ is not sufficient to reach b_d. For this purpose let $\varepsilon > 0$ be an arbitrary short but positive time span, we then have

$$\begin{aligned}
\mathrm{cf}(b_i, \mathrm{cf}^{-1}(b_i, b_d) - \varepsilon) &= \mathrm{cf}'(\mathrm{cf}'^{-1}(b_d) - \mathrm{cf}'^{-1}(b_i) - \varepsilon + \mathrm{cf}'^{-1}(b_i)) \\
&= \mathrm{cf}'(\mathrm{cf}'^{-1}(b_d) - \varepsilon) \;<\; b_d.
\end{aligned}$$

The reason for this is, that $\mathrm{cf}'^{-1}(b_d)$ is the minimal charging time ct, such that $\mathrm{cf}'(\mathrm{ct}) \geq b$ holds. Thus reducing the charging time by any $\varepsilon > 0$ will result in a final SoC less than b_d.

In our implementation we use piecewise linear functions to model all charging functions. A piecewise linear function is represented by an array of supporting points, sorted by charging time. Furthermore, we define that charging functions continue with a slope of 0 after the last supporting point. We now show how various types of charging functions can be modeled using this approach.

Regular Charging Stations. Recharging at a regular charging stations is typically a linear process until a SoC of about 80% is reached. Afterwards, the applied amperage is reduced in order to prevent the battery from getting damaged [JW06]. The overall charging speed depends furthermore on the battery's capacity and on the power available at the charging station. An example for such a charging function, where a SoC of 80% is reached after 4 time units, 90% is reached after 5 time units and the battery is fully charged after 7 time units would be represented by the supporting points $[(0,0),(4,80),(5,90),(7,100)]$. The associated charging function $\mathrm{cf}'(\mathrm{ct})$ is then given as

$$\mathrm{cf}'(\mathrm{ct}) := \begin{cases}
\frac{80}{4}b & \text{if } 0 \leq \mathrm{ct} < 4 \\
\frac{90-80}{5-4}b + 80 & \text{if } 4 \leq \mathrm{ct} < 5 \\
\frac{100-90}{7-5}b + 90 & \text{if } 5 \leq \mathrm{ct} < 7 \\
100 & \text{else.}
\end{cases}$$

Super Charger. Super chargers are a special kind of charging stations which feature a very high charging power. A drawback of these stations is, that the charging process has to be terminated when a SoC of 80% is reached, because it would otherwise damage the battery. So the charging function consists only of the linear part, and can thus be modeled using only two supporting points. The charging function cf' of a super charger that reaches a SoC of 80% after two time units would, for example, be represented by $[(0,0),(2,80)]$.

For such a function cf' it is important that we use the initial SoC as a lower bound for the resulting SoC after recharging in Definition 3.2. Because of this, the result of recharging with an initial SoC of 90% is $\mathrm{cf}(90, \mathrm{ct}) := \max(90, \mathrm{cf}'(\mathrm{ct} + \mathrm{cf}'^{-1}(90))) = \max(90, \mathrm{cf}'(\mathrm{ct})) = 90$ for any given charging time ct.

Battery Swapping Stations. Another type of "charging" stations are battery swapping stations (BSS), where the whole battery is exchanged with a new fully charged one. Modeling such a station using our model is a bit more complicated, because the charging process of a swapping station is not continuous as opposed to normal charging stations or super chargers. When using a battery swapping station, the SoC does not change at first (actually it drops to zero), before it after some time immediately increases to 100%.

If we want to model such a charging station using our model, we use the charging function cf' defined by $[(0, 100)]$. This function states that the battery is immediately fully charged when using the charging station. This is not very accurate so far, because swapping the battery will most certainly take some time. We model this time using additional edges in the graph.

Consider the scenario where vertex v represents a battery swapping station which needs one time unit to replace the battery. In this case we add a new vertex v' to the graph. This vertex represents the actual charging station using the charging function defined by $[(0, 100)]$. We then connect v' with the graph via two edges (v, v') and (v', v). Both of them have an energy consumption of 0. Furthermore we define their driving times as $\mathrm{dt}(v, v') = 1$ and $\mathrm{dt}(v', v) = 0$. By doing so, we obtain an accurate model for battery swapping stations, while the used charging function remains continuous.

The same approach can also be used to model additional effects which consume time when using a charging station. It is, for example, possible to model the time needed for parking the car, plugging in the power cable (and paying), using this approach.

4. Basic Approach

We now attempt to solve EVRC by using a modified version of the multi-objective shortest path search. Apart from having exponential worst case complexity, this yields a feasible base line algorithm for EVR. But the charging stations introduce a new kind of problem, in the sense that we have to choose a charging time for every charging station we use. The set of possible charging times is continuous and only limited by 0 as minimal charging time and by the time needed to recharge the complete battery as maximal charging time. Up to now Dijkstra's algorithm only had to choose from discrete sets of neighbor vertices. Therefore, it was possible to examine all possibilities.

Trying the same in order to find the optimal charging time means that we have to create a new label for every possible charging time. But this is not possible, considering the fact that the charging time has to be chosen from a continuous set. Using a discrete set of allowed charging times is an obvious workaround for this problem. In order to remain accurate, this set has to have a very fine resolution. But this would lead to a huge number of labels and therefore, the algorithm would be unpractical. Because of this we will try to restrict the set of meaningful charging times.

Obviously we do not want to waste travel time due to recharging. This means that the battery should not be charged any longer than necessary in order to reach the target. If we know the target vertex t, we could calculate the path from the current charging station to this target vertex. Afterwards, we know the amount of energy needed in order to use this path. We then could charge exactly this amount of energy. Unfortunately, this approach has two major problems.

- The first problem is that there is not only one path from the current charging station to t. Instead of this we have to consider a whole Pareto-set of paths. For example, it might not be the best solution to use the fastest path from a charging station to t, because this path might require a long charging time. If there exists a path with longer driving time, which requires significantly less energy, it might provide a lower travel time. Therefore, we have to compute complete Pareto-sets as we did before when solving EVR. Afterwards, it would be possible to calculate the charging time for every path contained in the Pareto-set. The optimal solution is then given by the path with minimal sum of driving time and charging time.
- The second problem is that we do not really know the target (or the location of the next stop) except for the last charging station used on the path to t. Consider

the case where one charging station is not sufficient in order to reach t. At some point during the computation we have to pick a charging time for the first charging station. When doing so, we would like to compute the Pareto-set of paths to the next stop, in order to determinate how much energy we have to charge. But now the next stop is not t but the second charging station, which is not yet known. An easy solution for this problem is to try every other charging station as second stop. Doing so would increase the number of labels exponentially with every additional charging stop needed. Therefore, this approach most probably would lead to an unpractical algorithm.

We now introduce a first practicable algorithm which simply ignores the first problem. For this algorithm we assume that we can always take the fastest path connecting two charging stations. Obviously such an algorithm cannot guarantee to find the optimal solution, however it will at least find feasible solutions.

4.1 Conventional Car Driver's Approach

We begin with a simple algorithm for calculating feasible paths. We search for a path similar to the way, a driver of a gas powered vehicle would do. This means to drive as fast as possible, or as one pleases, and to rely on the availability of filling stations wherever one needs to refuel.

Transferred to an algorithm naively this means to compute the shortest path regarding driving time without considering battery constraints and check whether this path is feasible afterwards. Of course it is unlikely that this path will contain sufficient charging stations. It is easy to see this by looking at highways, a charging station would not be located directly on the highway but on a resting area right next to it.

In order to take this into account, we again restrict our algorithm to use only the shortest path without considering battery constraints. But now this restriction must only be fulfilled between any two stops. With a stop being defined as starting at s, arriving at t, or using a charging station. An algorithm using this approach might be capable of finding a feasible path from s to t in a short time, but it is unlikely that such a path is optimal regarding travel time. Since recharging the battery can take a long time, it might be faster to use a slow path if this avoids the necessity of recharging or reduces the charging time sufficiently.

Because of this it is impossible that the conventional car drivers approach yields an optimal solution for our problem. Nevertheless its result may only be slightly slower than the optimal solution. Thus, we will recall this approach in Chapter 6.1 in order to develop an heuristic algorithm.

4.2 Charging Function Propagating Algorithm

Due to the additional assumption we made for the conventional car drivers approach, it was not possible to develop an optimal algorithm. Because of the fact that recharging can take a long time, it is important for an optimal algorithm to optimize both criteria, driving time and energy consumption. The algorithm we want to introduce now is based on the multi-objective shortest path search and solves the problem optimally. The algorithm maintains a label-set for each vertex in V. Each label contained in one of these sets consists of a travel time tt and a SoC b. As mentioned before, the main problem is settling vertices which represent charging stations, because it is not trivial to decide how much energy is needed to reach t or the next charging station. How much energy is needed depends on the further course of the path, but this information is not available at the time we have to settle a vertex representing a charging station.

The basic idea of our algorithm is to delay the decision about how much time should be spent at a charging station as long as possible. We achieve this by extending the labels. In addition to travel time and SoC, each label keeps a pointer to the last seen charging station on the associated path. As soon as we try to relax an edge with an energy consumes higher than the current label's SoC, we can charge exactly the missing energy retroactively.

So, for our new algorithm we want the labels to be a triple (tt, b, v) consisting of the travel time $\mathrm{tt} \in \mathbb{R}_{\geq 0}$, the current SoC $b \in \mathrm{B}$ and the last seen charging station $v \in \mathrm{CS} \cup \{\perp\}$, where $\perp$ means that we have not seen any charging station so far. We now want to discuss how this extension of the labels effects their dominance relation. The charging function associated with the last seen charging station v enables us to trade travel time for SoC. Thus, we have to consider all possible values, for the travel time needed to reach v, in order to define dominance.

One label dominates another only if it has a higher SoC for every possible travel time. A label $A = (\mathrm{tt}_A, b_A, u)$ dominates a label $B = (\mathrm{tt}_B, b_B, v)$ if and only if $\mathrm{tt}_A \leq \mathrm{tt}_B$ and there exists no charging time $\mathrm{ct} \geq 0$ such that $\mathrm{cf}_u(b_A, \mathrm{tt}_B - \mathrm{tt}_A + \mathrm{ct}) \leq \mathrm{cf}_v(b_B, \mathrm{ct})$ holds. This means that the minimal travel time of label B is greater than the minimal travel time of label A, and if we spent some time for recharging, so that the travel time provided by label A and B is equal, we will still arrive with a lower SoC when using label B.

4.2.1 SoC-Functions

In order to simplify the verification of dominance, we exploit the structure of charging functions. Instead of holding a triple consisting of tt and b together with a pointer to the last seen charging station, we now use a function $b \colon \mathbb{R}_{\geq 0} \to \mathrm{B} \cup \{-\infty\}$ which maps travel time to SoC. The special value $-\infty$ indicates that it is not possible to traverse the path associated with the label in the given travel time. We restrict ourselves to concave piecewise linear functions represented by a sorted vector of supporting points, which is possible since all charging functions we use are piecewise linear.

Let $[(\mathrm{tt}_1, b_1), \ldots, (\mathrm{tt}_k, b_k)]$ be a sequence of supporting points sorted by travel time in ascending order as label for the vertex $v \in V$. We define the SoC-function $b(\cdot)$ associated with this sequence as

$$b(t) := \begin{cases} -\infty & \text{if } 0 \leq t < \mathrm{tt}_1 \\ \frac{(t-\mathrm{tt}_1)(b_2-b_1)}{\mathrm{tt}_2} + b_1 & \text{if } \mathrm{tt}_1 \leq t < \mathrm{tt}_2 \\ \vdots & \\ \frac{(t-\mathrm{tt}_{k-1})(b_k-b_{k-1})}{\mathrm{tt}_k} + b_{k-1} & \text{if } \mathrm{tt}_{k-1} \leq t < \mathrm{tt}_k \\ b_k & \text{else.} \end{cases}$$

An example figure for such a SoC-Function with supporting points $[(1, 1), (3, 4), (6, 7)]$ is shown in Figure 4.1.

Given such a function, we see that it is impossible to arrive at vertex v in less than tt_1 time, indicated by the value $-\infty$. We call tt_1 the *earliest arrival time* for vertex v. For any travel time t between tt_1 and tt_k we choose $1 < i \leq k$ with $\mathrm{tt}_{i-1} \leq t < \mathrm{tt}_i$ and interpolate linearly between the two points $(\mathrm{tt}_{i-1}, b_{i-1})$ and (tt_i, b_i). For any travel time t greater than tt_k there exists no Pareto-optimal solution. However, we allow to simply wait at v until the time has passed, during which the SoC does not change. If a SoC-function is equal to $-\infty$ for every value of t, the associated path is unfeasible, thus it is unfeasible to reach v using this path.

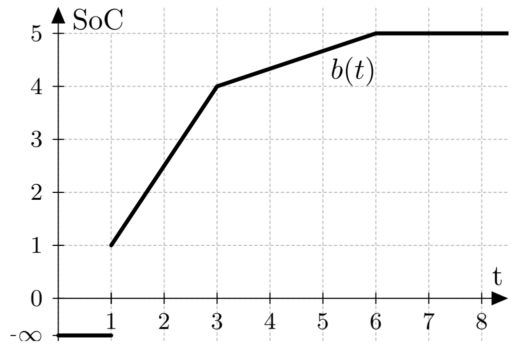

Figure 4.1: An example for a SoC-Function $b(t)$ with supporting points $[(1,1),(3,4),(6,7)]$.

Dominance. Using SoC-functions as labels simplifies the definition of dominance considerably. The SoC-function $b_A(\cdot)$ dominates the function $b_B(\cdot)$ (denoted by $b_A \propto b_B$) if and only if for each $t \in \mathbb{R}_{\geq 0}$ the condition $b_A(t) \geq b_B(t)$ holds. Since we keep the supporting points of every SoC-function ordered by travel time, this can be checked efficiently by performing a linear sweep over the supporting points of both functions.

Lemma 4.1. *Let $b_A(\cdot)$ and $b_B(\cdot)$ be two piecewise linear SoC-Functions defined by the supporting points $[(\mathrm{tt}_1^A, b_1^A), \ldots, (\mathrm{tt}_k^A, b_k^A)]$ and $[(\mathrm{tt}_1^B, b_1^B), \ldots, (\mathrm{tt}_\ell^B, b_\ell^B)]$. The function b_A dominates b_B if $b_A(t) \geq b_B(t)$ holds for every $t \in T$, where T is the set containing the travel times of all supporting points, defined as $T := \{\mathrm{tt}_1^A, \ldots, \mathrm{tt}_k^A, \mathrm{tt}_1^B, \ldots, \mathrm{tt}_\ell^B\}$.*

Proof. Let tt_i and tt_j be two consecutive elements of T, which means there exists no $\mathrm{tt}_c \in T$ such that $\mathrm{tt}_i < \mathrm{tt}_k < \mathrm{tt}_j$. This means that both functions $b_A(\cdot)$ and $b_B(\cdot)$ are linear between tt_i and tt_j. The precondition that $b_A(\mathrm{tt}_i) \geq b_B(\mathrm{tt}_i)$ and $b_A(\mathrm{tt}_j) \geq b_B(\mathrm{tt}_j)$ holds implies that $b_A(t) \geq b_B(t)$ holds for all $\mathrm{tt}_i \leq t \leq \mathrm{tt}_j$. □

If both SoC-functions $b_A(\cdot)$ and $b_B(\cdot)$ are monotonously increasing, which is the case if all participating charging functions are feasible, then there exists an additional way of verifying dominance. Instead of comparing the resulting SoC for the same travel time, it is possible to compare the travel time required to reach a certain SoC.

Lemma 4.2. *Let $b_A(\cdot)$ and $b_B(\cdot)$ be two monotonously increasing SoC-Functions. In this case b_B is dominated by b_A ($b_A \propto b_B$) if there exists an $\Delta \geq 0$ for every travel time $\mathrm{tt} \geq 0$, such that $b_A(\mathrm{tt}) \geq b_b(\mathrm{tt} + \Delta)$.*

Proof. The lemma follows directly from the definition of monotonous increasing functions. Since b_B is monotonously increasing $\mathrm{tt} \leq \mathrm{tt}'$ implies that $b_B(\mathrm{tt}) \leq b_B(\mathrm{tt}')$ holds. Therefore, we have $b_A(\mathrm{tt}) \geq b_B(\mathrm{tt} + \Delta) \geq b_B(\mathrm{tt})$ for every $\mathrm{tt} \geq 0$ and given $\Delta \geq 0$, which proves the claim. □

As before this statement of dominance can easily be verified be looking at all supporting points of b_A and b_B, provided that both functions are piecewise linear.

We now describe how the new SoC-functions can be used as labels for the multi-objective shortest path search in order to solve EVRC.

4.2.2 Pareto Search for SoC-Functions

During the initialization phase we do not need to change much. As before, all label-sets are initialized as the empty set. Afterwards, the initial label defined by $[(0, b(s))]$ is inserted into the label-set for the vertex s and s itself is inserted into the queue.

Vertex Settling. After the initialization phase the algorithm continues with the main loop. Here it settles the minimal label contained in Q until Q is empty. To enable this behavior for our SoC-functions, we need to define an order on them. We also want to keep the label-setting property of Dijkstra's algorithm, so we have to define the order accordingly. Fur this purpose, we order SoC-functions by their earliest possible arrival time, i.e., the travel time of the first supporting point. Furthermore, we use the SoC of the first supporting point to break ties. By doing so we ensure that the label-setting property holds, in the sense, that a settled label will never be dominated. Since our graph contains only edges with positive travel time, the earliest arrival time will never decrease due to an edge relaxation. Therefore, it is impossible that a SoC-Function gets dominated, once it was settled.

Edge Relaxations. Next, we show how SoC-functions are handled during an edge relaxation. Consider the label $A = [(\mathrm{tt}_1, b_1), \ldots, (\mathrm{tt}_k, b_k)]$ of u being settled and the according SoC-function $b_A(\cdot)$. Relaxing the edge $e = (u, v) \in E$ generates a new label B for v. The SoC-function $b_B(t)$ for this label has to reflect that an additional time of $\mathrm{dt}(e)$ is needed to reach v and that traversing the edge consumes $\mathrm{cons}(e)$ energy. This can be achieved by shifting the function $b_A(\cdot)$ by $(\mathrm{dt}(e), -\mathrm{cons}(e))$. In addition to that, we have to ensure that $b_B(t) \in \mathrm{B} \cup \{-\infty\}$ holds for every value of t. Thus we define the result $b_B(\cdot)$ of relaxing the edge e, which is denoted by $b_A \circ e$, as

$$(b_A \circ e)(t) = b_B(t) := \begin{cases} -\infty & \text{if } b'_B(t) < 0 \\ M & \text{if } b'_B(t) > M \\ b'_B(t) & \text{else} \end{cases}$$

where $b'_B(t) := b_A(t - \mathrm{dt}(e)) - \mathrm{cons}(e)$ is the shifted function. The definition of $b_B(\cdot)$ reflects, that arriving at v with negative SoC renders the path unfeasible, thus $b_B(\cdot)$ is set to $-\infty$ in this case. On the other hand, a SoC greater than M means that overcharging occurs, hence these values are limited to M.

Up to now we have defined all operations needed, to use SoC-functions together with the multi-objective shortest path search. In fact, the introduced modifications are sufficient to solve EVR. But we have not used the charging stations so far. Now we will integrate them into the algorithm.

Charging Station Settling. Every time we attempt to settle a label of a vertex v, we check whether v is a charging station. If that is the case we have to explore the possibility of recharging at this station. Here, we distinguish two different cases. One where the label only contains one supporting point and another where the label contains more than one supporting point.

A label $[(\mathrm{tt}_1, b_1)]$ consisting of only one supporting point can occur in two situations. Either the associated path does not contain any charging station up to now, or the path contains some charging stations but it exists only one possible charging time in order to reach v. In both cases the path from s to v is distinct, which makes it easy to integrate the new charging station, since it cannot be a disadvantage to use the new charging station. Before settling the label we simply change its SoC-function to

$$b(t) := \begin{cases} -\infty & \text{if } t < \mathrm{tt}_1 \\ \mathrm{cf}_v(b_1, t - \mathrm{tt}_1) & \text{else.} \end{cases}$$

This new function for the label states, that we can start recharging as soon as we arrive at v. The supporting points needed to describe the changed function $b(\cdot)$ can be obtained by shifting the supporting points of cf_v.

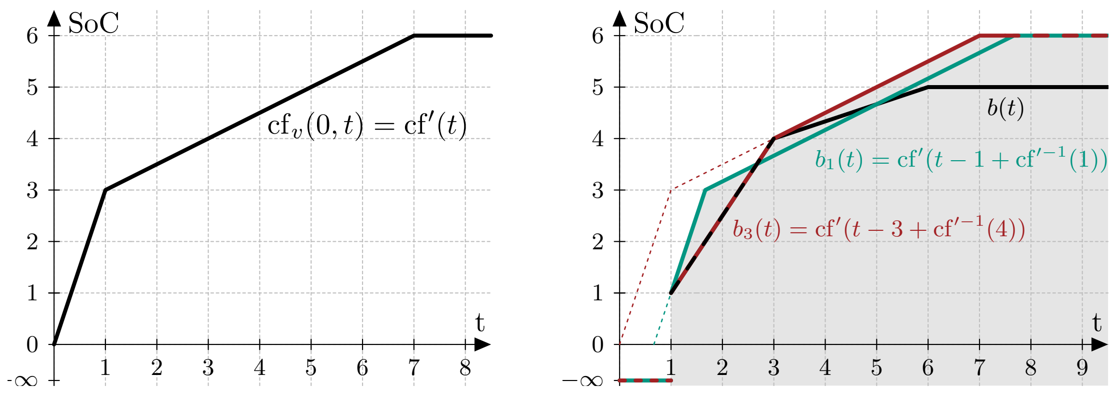

Figure 4.2: An example showing the results of switching the last seen charging station. The previous SoC-function $b(t)$ is defined by $[(1,1),(3,4),(6,5)]$. The new charging function $\mathrm{cf}_v(0,t)$ is defined by $[(0,0),(1,3),(7,6)]$. The shaded area contains all points dominated by some $b_{\mathrm{tt}}(t)$. Yet the complete shaded area can be dominated by using only the functions $b_1(t)$ and $b_3(t)$.

Settling a charging station gets more complicated if the current label $[(\mathrm{tt}_1, b_1), \ldots, (\mathrm{tt}_k, b_k)]$ contains more than one supporting point. In this case we have to decide how much energy should be charged using the last seen charging station, before we switch over to the new one. But this depends on the energy needed for the remaining path from v to the target vertex, which we do not know yet. Given the SoC-function $b(\cdot)$ associated with the current label, we can define a new function $b_{\mathrm{tt}}(t)$ for every $\mathrm{tt} \geq \mathrm{tt}_1$

$$b_{\mathrm{tt}}(t) := \begin{cases} b(t) & \text{if } t < \mathrm{tt} \\ \mathrm{cf}_v(b(\mathrm{tt}), t - \mathrm{tt}) & \text{else.} \end{cases}$$

This function is a correct SoC-Function for the case that it took a time of tt to arrive at v (including recharging at the last seen charging station), where we switch over to the new charging station. Keeping all of these functions does not lead to a practicable algorithm. Luckily, it is possible to choose a small set of these functions such that all the other functions are dominated by them. The reason for this is, that our SoC-functions as well as the charging function are piecewise linear and concave.

Switching the charging station makes only sense, if the charging speed offered by the new charging station is higher than the charging speed at the last seen station, for the particular SoC at which the station is changed. Furthermore, if this situation occurs, it is the best choice to switch over to the new charging station as soon as it offers a higher charging speed. There are, however, only two possible reasons for this. Either the charging speed of the new station suddenly increases and surpasses the speed of the old one, or the charging speed available at the old station decreases. The first case cannot occur since our charging functions are concave, meaning that the charging speed will only decrease as the charging proceeds. Thus, we are left with the second case, i.e., the slope of the current SoC-function decreases. Since our SoC-functions are piecewise linear this can only occur at a supporting point of the function.

An example for this situation is shown in Figure 4.2. This example also shows, how the charging function $\mathrm{cf}(0,t)$ gets shifted sideways when defining $b_{\mathrm{tt}}(t)$ so that it intersects the SoC-function $b(t)$ at exactly tt. Thus, we only need to identify those functions, which are not dominated by any other function, and can proceed with them. See Figure 4.2 for an example of this situation.

Theorem 4.3. *Let $b(t)$ be a SoC-Function defined by the sequence $[(\mathrm{tt}_1, b_1), \dots, (\mathrm{tt}_k, b_k)]$ of supporting points. Furthermore let $\mathrm{cf}(b, \mathrm{ct})$ be a feasible charging function which is defined by $\mathrm{cf}'(\mathrm{ct})$ according to Definition 3.2 on page 25. Then for every $\mathrm{tt} \geq \mathrm{tt}_1$, there exists an $i \in [1, k]$ such that $b_{\mathrm{tt}_i} \propto b_{\mathrm{tt}}$, i.e. it is not worse to switch over to the new charging station at tt_i than at tt.*

Proof. We prove this theorem in two steps. First, we show that $b_{\mathrm{tt}_k} \propto b_{\mathrm{tt}}$ holds for the special case of $\mathrm{tt} \geq \mathrm{tt}_k$. Afterwards, we prove the claim for the more general case of tt lying between two supporting points, i.e., $\mathrm{tt}_1 \leq \mathrm{tt} \leq \mathrm{tt}_k$.

Case 1. We start, by showing that b_{tt_k} dominates b_{tt} for the special case of $\mathrm{tt} \geq \mathrm{tt}_k$. This is only valid if $b_{\mathrm{tt}_k}(t) \geq b_{\mathrm{tt}}(t)$ holds for every $t \geq 0$. For $t < \mathrm{tt}_k$ both SoC-functions, $b_{\mathrm{tt}_k}(t)$ and $b_{\mathrm{tt}}(t)$, are equal to $b(t)$ per definition, thus $b_{\mathrm{tt}_k}(t) \geq b_{\mathrm{tt}}(t)$ holds. For $t \geq \mathrm{tt}_k$ we have

$$b_{\mathrm{tt}_k}(t) := \mathrm{cf}(b(\mathrm{tt}_k), t - \mathrm{tt}_k) = \mathrm{cf}(b(\mathrm{tt}), t - \mathrm{tt}_k),$$

because $b(\cdot)$ is constant for values greater or equal to tt_k. If additionally $t < \mathrm{tt}$ holds, then $b_{\mathrm{tt}}(t)$ is defined as $b(\mathrm{tt})$, which is less or equal to $\mathrm{cf}(b(\mathrm{tt}), t - \mathrm{tt}_k)$, since recharging never decreases the SoC. If otherwise $t \geq \mathrm{tt}$ holds, then $b_{\mathrm{tt}}(t)$ is defined as $\mathrm{cf}(b(\mathrm{tt}), t - \mathrm{tt})$. This is again less or equal to $\mathrm{cf}(b(\mathrm{tt}), t - \mathrm{tt}_k)$ because $t - \mathrm{tt} \leq t - \mathrm{tt}_k$ and cf is monotonous. Therefore, $b_{\mathrm{tt}_k} \propto b_{\mathrm{tt}}$ holds for $\mathrm{tt} \geq \mathrm{tt}_k$.

Case 2a. We now address the second case, that is $\mathrm{tt}_1 \leq \mathrm{tt} \leq \mathrm{tt}_k$. In this case there exists an index $i \in [1, k-1]$, such that $\mathrm{tt}_i \leq \mathrm{tt} \leq \mathrm{tt}_{i+1}$ holds. We now show that either b_{tt_i} or $b_{\mathrm{tt}_{i+1}}$ dominates b_{tt}. Which of these functions dominates b_{tt} is determined by the slope of b and cf at the point were we switch over to the new charging function. We first consider the case of

$$\frac{\partial b(t)}{\partial t}(\mathrm{tt}) = \frac{b_{i+1} - b_i}{\mathrm{tt}_{i+1} - \mathrm{tt}_i} < \frac{\partial \mathrm{cf}(b(\mathrm{tt}), t - \mathrm{tt})}{\partial t}(\mathrm{tt}).$$

This means that the new charging station offers a higher charging speed than the last seen one for a SoC of $b(\mathrm{tt})$. Thus the new charging function should be used in order to benefit from the higher charging speed. Since b is piecewise linear and cf is concave this is still valid if we switch over to the new charging station earlier. Thus $b_{\mathrm{tt}_i} \propto b_{\mathrm{tt}}$ holds in this case, which we now prove formally. As before, both SoC-functions are per definition equivalent for $t < \mathrm{tt}_i$ and therefore $b_{\mathrm{tt}_i}(t) \geq b_{\mathrm{tt}}(t)$ holds. For $t \geq \mathrm{tt}_i$ we use that, changing time at which we switch over to the new charging function, from tt to tt_i, is equivalent to shifting the charging function regarding travel time. This means there exists a value Δ such that $\mathrm{cf}(b(\mathrm{tt}_i), t - \mathrm{tt}_i)$ is equivalent to $\mathrm{cf}(b(\mathrm{tt}), t + \Delta - \mathrm{tt})$ (See Figure 4.3a for an example). This value Δ is given by $\mathrm{tt} - \mathrm{tt}_i - \mathrm{cf}^{-1}(b(\mathrm{tt}_i), b(\mathrm{tt}))$:

$$\begin{aligned}
\mathrm{cf}\left(b(\mathrm{tt}), t + \Delta - \mathrm{tt}\right) &= \mathrm{cf}\left(b(\mathrm{tt}), t + \mathrm{tt} - \mathrm{tt}_i - \mathrm{cf}^{-1}(b(\mathrm{tt}_i), b(\mathrm{tt})) - \mathrm{tt}\right) \\
&= \mathrm{cf}\left(b(\mathrm{tt}), t - \mathrm{tt}_i - \mathrm{cf}^{-1}(b(\mathrm{tt}_i), b(\mathrm{tt}))\right) \\
&= \mathrm{cf}\left(b(\mathrm{tt}), t - \mathrm{tt}_i + \mathrm{cf}'^{-1}(b(\mathrm{tt}_i)) - \mathrm{cf}'^{-1}(b(\mathrm{tt}))\right) \\
&= \mathrm{cf}'\left(t - \mathrm{tt}_i + \mathrm{cf}'^{-1}(b(\mathrm{tt}_i))\right) - \mathrm{cf}'^{-1}\left(b(\mathrm{tt})\right) + \mathrm{cf}'^{-1}\left(b(\mathrm{tt})\right) \\
&= \mathrm{cf}'\left(t - \mathrm{tt}_i + \mathrm{cf}'^{-1}(b(\mathrm{tt}_i))\right) \\
&= \mathrm{cf}\left(b(\mathrm{tt}_i), t - \mathrm{tt}_i\right)
\end{aligned}$$

Furthermore Δ is not negative, since $\mathrm{tt} - \mathrm{tt}_i \geq \mathrm{cf}^{-1}(b(\mathrm{tt}_i), b(\mathrm{tt}))$ holds. The reason for this is that $\mathrm{tt} - \mathrm{tt}_i$ is the time needed to charge from $b(\mathrm{tt}_i)$ to $b(\mathrm{tt})$ using the last charging function and $\mathrm{cf}^{-1}(b(\mathrm{tt}_i), b(\mathrm{tt}))$ is the time needed, using the new charging function. As the

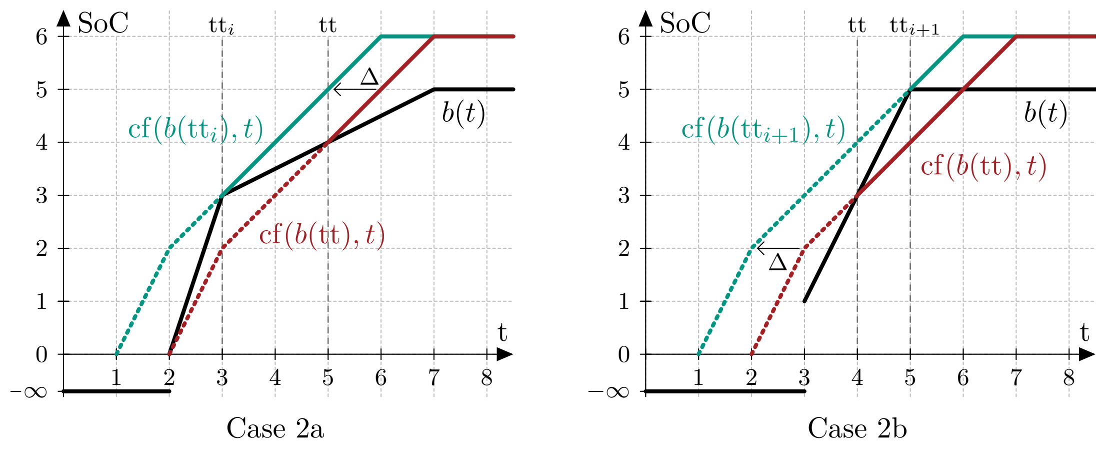

Figure 4.3: Examples for case 2a and 2b of the proof. The charging function is defined by $[(0,0),(1,2),(5,6)]$ in both cases. For case 2a the current SoC-function $b(t)$ is given as $[(2,0),(3,3),(7,5)]$, so that the slope of $b(t)$ for $t = \text{tt}$ is less than the slope of $\text{cf}(b(\text{tt}), t)$ for $t = \text{tt}$. For case 2b the current SoC-function $b(t)$ is given as $[(3,1),(5,5)]$, thus the slope of $b(t)$ for $t = \text{tt}$ is greater than the slope of $\text{cf}(b(\text{tt}), t)$ for $t = \text{tt}$. In both cases the charging function can be shifted to the left by Δ, such that it intersects $b(t)$ at a supporting point.

slope of the old charging function, is less than the slope of the new charging function, the new charging function needs less time to charge from $b(\text{tt}_i)$ to $b(\text{tt})$.

Since $\Delta \geq 0$ holds, we can prove $b_{\text{tt}_i}(t) \geq b_{\text{tt}}(t)$ for $t \geq \text{tt}$ quite easily by using Lemma 4.2

$$b_{\text{tt}_i}(t) = \text{cf}(b(\text{tt}_i), t - \text{tt}_i) = \text{cf}(b(\text{tt}), t + \Delta - \text{tt}) \geq \text{cf}(b(\text{tt}), t - \text{tt}) = b_{\text{tt}}(t)$$

In particular this equation shows, that $b_{\text{tt}_i}(\text{tt}) \geq b_{\text{tt}}(\text{tt})$ holds. We furthermore know that $b_{\text{tt}_i}(\text{tt}_i) = b_{\text{tt}}(\text{tt}_i)$ holds per definition, $\text{cf}(b,t)$ is concave and $b(t)$ is linear in the interval $[\text{tt}_i, \text{tt}]$. Therefore, $b_{\text{tt}_i}(t) \geq b_{\text{tt}}(t)$ also holds for $t \in [\text{tt}_i, \text{tt}]$, which proves $b_{\text{tt}_k} \propto b_{\text{tt}}$.

Case 2b. The last case we have to address is the one that the last seen charging station offers a charging speed at least as high as the new one for a SoC of $b(\text{tt})$, or more formally

$$\frac{\partial b(t)}{\partial t}(\text{tt}) \geq \frac{\partial \text{cf}(b(\text{tt}), t - \text{tt})}{\partial t}(\text{tt}).$$

An example for this case is shown in Figure 4.3b. It is clear that switching over to the new charging station is not reasonable, since the current one offers a higher charging speed. The earliest possible moment at which the new charging function could offer a higher charging speed than the current station, is the next supporting point of $b(t)$. Thus $b_{\text{tt}_{i+1}} \propto b_{\text{tt}}$ holds in this case, which can be proven similar to case 2a.

In this case, Δ is given by $\text{cf}^{-1}(b(\text{tt}), b(\text{tt}_{i+1})) - (\text{tt}_{i+1} - \text{tt})$, which is the difference of the time needed for the new charging station to charge from $b(\text{tt})$ to $b(\text{tt}_{i+1})$ and the time needed by the current station to do so. Since the current charging station offers a charging speed at least as high as the new one, this difference, and therefore Δ is again not negative. Using the new Δ we can show (along the lines of case 2a) that

$$\text{cf}\left(b(\text{tt}), t + \Delta - \text{tt}\right) = \text{cf}\left(b(\text{tt}_i), t - \text{tt}_{i+1}\right)$$

holds. From there on the proof is analogous to case 2a. □

Whenever our algorithm extracts a vertex from the queue that represents a charging station, we can apply Theorem 4.3. Since we know that changing the charging station is only meaningful at a supporting point of the current SoC-function, we simply create one new label for each supporting point in order to explore the possibility of switching over to the new charging station at that point.

The algorithm terminates as soon as a label for the target vertex t gets settled. In this case the earliest arrival time of this label is the shortest possible travel time of any path from s to t. Furthermore, the path associated with this label is the shortest feasible s-t-path.

Path Retrieval. If we are not only interested in the shortest possible travel time, but also want to know the shortest path as well as the charging time for every used charging station, we have to extend the labels by a parent pointer. When doing so, the shortest path can be found by backtracking these pointers from t to s. During the backtracking process we keep track of the arrival time and the SoC for every vertex contained in the path. For the vertex t these values are simply given by the first supporting point of the label for t. Backtracking an edge e reduces the arrival time by $\mathrm{dt}(e)$ and increases the SoC by $\mathrm{cons}(e)$. If we reach a charging station these values can be used in order to calculate the charging time.

4.2.3 Implementation Details

Whenever an edge $e = (u, v)$ is relaxed, we have to duplicate and modify the SoC-function associated with u. As it has been described in the high level algorithm up to know, this takes at least linear time in the size of the SoC-function, i.e., its number of supporting points. But the modification actually performed in order to create the new label for v is fairly simple. The function gets only shifted by $\mathrm{dt}(e)$ and $\mathrm{cons}(e)$. If this shift causes the SoC-function to exceed M, which represents over-charging, The function is simply truncated. Under-charging is handled similarly.

We can optimize the relax operation so that only constant time is needed. In order to do so we do not apply the shift to the function but rather accumulate it. Thus when relaxing an edge, it is not necessary to shift each of the SoC-function supporting points, but only the overall shift of the function has to be adjusted. In addition to shifting the SoC-function we also must account for over- and under-charging. Therefore, we have to keep track of additional values specifying the maximal reachable SoC and the minimal needed SoC. But this is exactly what is done by the energy consumption functions introduced in Chapter 2.3.3. So our final SoC-function label consists of the quadruple $(v, b, \mathrm{c}, \mathrm{tt})$:

- The vertex v of the last seen charging station along the route to the current vertex. This vertex is used as a pointer to the charging function $\mathrm{cf}_v(b, \mathrm{ct})$, which provides the supporting points of the b-function.
- The SoC b at which the last seen charging station was reached. Inserting this into the charging function of vertex v, we obtain a one-dimensional function mapping time to SoC. This defines the basic shape of the to be shifted SoC function.
- An energy consumption function c that describes the energy consumption along the used path from v to the current vertex. cost_c is used to shift the basic SoC function, while $\mathrm{minIn}_\mathrm{c}$ and $\mathrm{maxOut}_\mathrm{c}$ are used to truncate the SoC-function according to over- and under-charging.
- The minimal travel time tt required to reach the current vertex. This value is also applied as a shift to the basic SoC-function.

Relaxing an edge e can indeed be done in constant time using these SoC-function labels. The values v and b are simply copied to the new label. The energy consumption function c is linked with the energy consumption of e and the travel time tt is increased by the driving time of e. Similarly switching over to a new charging station can be done in constant time. We do so by assigning the new charging station to v. The values b and tt are set to the time and SoC at which we switch over to the new charging station. Finally, the energy consumption function c is cleared so that it represents an energy consumption of 0.

While relaxing an edge and swapping the last seen charging station can be done quite easily, it becomes a bit more complicated to evaluate the SoC function. Previously the value $b(t)$ of the charging function could be obtained by linear interpolation of the according supporting points. Using the quadruple representation of the SoC-function, its value is now defined as $b(t) := b - \mathrm{c}(\mathrm{cf}_v(b, t - \mathrm{tt}))$.

Complexity. The complexity of the labels used in this algorithm is bound by the number of supporting points used to define the charging functions. Furthermore we do use neither less nor more labels than the plain version of Label Setting Dijkstra. Therefore, the worst case complexity is still exponential in the size of the graph.

5. Speedup Techniques

We now want to utilize known speedup techniques in order to improve our charging station aware algorithm. Therefore, we analyze how A* and Contraction Hierarchies can be adapted to our new scenario. Finally, we show a combination of both techniques as it is used by our final algorithm.

5.1 A* Search

First we take a look at the A* technique. The main challenge of this technique is finding a good admissible potential function, which estimates the distance from every vertex to t. Common approaches make use of the triangle inequality or landmarks in order to obtain a good potential for the shortest path problem [GH05]. The special structure of EVRC, however, enables us to develop new, specialized potentials. The initial idea here is, that we can drop some of the constraints. This produces a simpler problem, which can be solved using Dijkstra's algorithm. Afterwards, the resulting potential can be used to speedup the original query.

We start by dropping all battery constraints, leaving us with the normal shortest path problem for the driving time metric. Given a target t, we can perform a backwards run of Dijkstra's algorithm starting at t. As result we get the unconstrained minimal driving time from every vertex to t. Since adding battery constraints reduces the number of feasible paths, they can only lead to an increased driving time. Therefore, the unconstrained driving time is an lower bound for the constrained driving time and can be used as potential for the driving time of the constrained problem. Thus, we can use it to obtain a first speedup technique for EVRC.

Now we want to do better and try to incorporate at least some information about energy consumption. Just as we can use driving time as metric for Dijkstra's algorithm we can use energy consumption. By doing so we obtain for every vertex the minimal energy required to reach t. If for some vertex $u \in V$ this minimal energy exceeds the available energy at u, then it is not possible to reach t from u without recharging. Furthermore it is evident that, on any feasible path from u to t, at least the difference between minimal required energy and available energy has to be obtained using a charging station.

Our first observation here is, that the minimal travel time for reaching t does not only depend on the vertex u where the path begins, but also on the SoC at this vertex u. So in the context of EVRC we would like to have a potential $\pi_t \colon V \times \mathrm{B} \to \mathbb{R}_{\geq 0}$, which assigns a lower bound on the travel time to every pair of a vertex v and a SoC b.

A second observation is that we can calculate a lower bound for the amount of energy that has to be charged at a charging station. But when calculating a potential function we are interested in a lower bound for travel time. Thus we need a way to convert energy into travel time. Luckily this is exactly what the charging function, or more precise the inverse of the charging function, does. A problem with this approach is that we do not know which charging station will be used by the optimal solution. We also do not know the SoC with which the charging station is reached. In order to solve this problem we will simply assume that it is always possible to use the fastest charging station.

We will now use both observations in order to calculate a new, more precise potential function for the minimal travel time.

5.1.1 Uniform Charging Speed

As before our algorithm starts with a backwards execution of Dijkstra's algorithm using the driving time metric. Afterwards, a second backwards execution using energy consumption as metric is performed. By doing so we obtain a lower bound $\pi_{\mathrm{dt}} \colon V \to \mathbb{R}$ for the driving time from every vertex to t and a lower bound $\pi_{\mathrm{cons}} \colon V \to \mathbb{R}$ for the energy consumption from every vertex to t. Finally we calculate an upper bound for the charging speed $\mathrm{cf}_{\max}$

$$\mathrm{cf}_{\max} := \max \left[\frac{\partial \mathrm{cf}_v(b,t)}{\partial t}(\mathrm{ct}) \right] \quad \text{s.t.} \quad v \in \mathrm{CS}, b \in \mathrm{B}, \mathrm{ct} \in \mathbb{R}_{\geq 0}.$$

So the maximal charging speed is given by the maximal slope of any charging function cf_v for any SoC b and charging time ct. Since all charging functions are given as piecewise linear function, this value can be obtained easily by a linear sweep over all supporting points.

Given π_{dt}, π_{cons} and $\mathrm{cf}_{\max}$ we can define an improved potential $\pi_1 \colon V \times \mathrm{B} \to \mathbb{R}_{\geq 0}$ for manipulating the order in which the labels of our basic algorithm from Section 4.2 get settled

$$\pi_1(v,b) := \begin{cases} \pi_{\mathrm{dt}}(v) & \text{if } b \geq \pi_{\mathrm{cons}}(v) \\ \pi_{\mathrm{dt}}(v) + \frac{\pi_{\mathrm{cons}}(v) - b}{\mathrm{cf}_{\max}} & \text{else.} \end{cases}$$

Given a label $A = [(\mathrm{tt}_1, b_1), \ldots, (\mathrm{tt}_k, b_k)]$ for vertex v, the value $\pi_1(v, b_1)$ is a lower bound for the remaining travel time from v to t. This holds because for $b \geq \pi_{\mathrm{cons}}(v)$ the value $\pi_1(v, b_1)$ is simply the minimal driving time from v to t without constraints. In the case of $b < \pi_{\mathrm{cons}}(v)$ the target t is not reachable from v without recharging and it is not possible to recharge the missing energy in less than $(\pi_{\mathrm{cons}}(v) - b) / \mathrm{cf}_{\max}$ time. Furthermore it is sufficient to use the label's first supporting point, although the label describes an entire function $b_A(\cdot)$ mapping arrival time to SoC. The reason for this is that $b_A(\cdot)$ is only a shifted charging function. Therefore, its slope cannot be greater than $\mathrm{cf}_{\max}$ which means that the sum of arrival time at v and $\pi(v, \mathrm{tt}_i)$ is minimal for the first supporting point:

$$\forall_{1 < i \leq k} : \quad \mathrm{tt}_1 + \pi_1(v, b_1) \ \leq \ \mathrm{tt}_i + \pi_1(v, b_i).$$

While $\pi_1(v, b)$ is certainly a better potential than the minimal driving time without constraints, it is still a rough estimation. One problem is, that we use the minimal driving time as lower bound as soon as the SoC is sufficient for the most energy efficient path. But in reality the driving time of these both paths may differ significantly. This results in a huge difference between the calculated lower bound and the actual travel time, especially if the target is not reachable without recharging. In this case we assume the lower driving time for the fastest path without considering the associated higher energy consumption and therefore longer charging time.

5.1.2 Optimal Energy Consumption Rate

Now we develop a potential function which considers the trade-off between driving time and energy consumption. The new potential function only affects the case where we have to use a charging station.

We invest some time in order to receive energy, every time a charging station is used. Thus, wasting energy on a path that makes use of a charging station is equivalent to wasting time. In order to find the optimal path i.e., a path that wastes neither time nor energy, we have to look at the combined time for driving and charging.

We define a new edge weight $\omega\colon E \to \mathbb{R}$ which is a lower bound on the travel time of an edge if the energy consumed on this edge has to be obtained using a charging station. Since the charging rate of all charging stations is limited by $\mathrm{cf}_{\max}$, we define $\omega(\cdot)$ as

$$\omega(e) := \mathrm{dt}(e) + \frac{\mathrm{cons}(e)}{\mathrm{cf}_{\max}}.$$

Note that ω might assign negative values as lower bound for travel time, if the energy consumption $\mathrm{cons}(e)$ is negative. This is fine because an edge with negative consumption provides energy that otherwise must have been charged. We still ignore the fact that energy can be lost due to over-charging. Because of this we can use ω to calculate a lower bound on the driving and recharging time of a whole path $P = (u_1, \ldots, u_k)$. Given that we start at vertex u_1 with an empty battery, we can be sure that it takes at least $\omega(P)$ time to reach u_k. If we start at vertex u_1 with some SoC $b(u_1) > 0$, we might overestimate the total travel time for the path P when using $\omega(P)$ as potential. We solve this by subtracting the time that would have been needed to charge the battery up to $b(u_1)$.

To obtain our new potential function, we perform a third backwards execution of Dijkstra's algorithm starting at t and using ω as metric. This yields a lower bound $\pi_{\mathrm{tt}}\colon V \to \mathbb{R}$ for the combined driving and charging time. Together with the lower bounds π_{dt} and π_{cons} from the previous subsection we can define a new potential

$$\pi_2(v, b) := \begin{cases} \pi_{\mathrm{dt}}(v) & \text{if } b \geq \pi_{\mathrm{cons}}(v) \\ \pi_{\mathrm{tt}}(v) - \frac{b}{\mathrm{cf}_{\max}} & \text{else.} \end{cases}$$

As before it is clear that the potential is admissible for the case $b \geq \pi_{\mathrm{cons}}(v)$. We now assume that it is not possible to reach t from v with SoC b i.e., $b < \pi_{\mathrm{cons}}(v)$. Furthermore we assume that there is a feasible path P from v to t such that its total travel time is less than $\pi_2(v, b)$. A lower bound minIn for the energy that is needed to drive along P is given by $\mathrm{cons}(P)$. If we subtract b from this value we get a lower bound for the energy that has to be charged. Thus a lower bound for the total travel time on P is given by

$$\begin{aligned} \mathrm{tt}(P) + \frac{\mathrm{cons}(P) - b}{\mathrm{cf}_{\max}} &= \sum_{e \in P} \left(\mathrm{dt}(e) + \frac{\mathrm{cons}(e)}{\mathrm{cf}_{\max}} \right) - \frac{b}{\mathrm{cf}_{\max}} \\ &= \sum_{e \in P} \omega(e) - \frac{b}{\mathrm{cf}_{\max}} \\ &= \omega(P) - \frac{b}{\mathrm{cf}_{\max}} \end{aligned}$$

But $\pi_{\mathrm{tt}}(v)$ is a lower bound for $\omega(P)$. Therefore, $\pi_2(v, b)$ has to be less than $\omega(P) - \frac{b}{\mathrm{cf}_{\max}}$, which is a contradiction to the assumption that the total travel time for the path P is less than $\pi_2(v, b)$.

Note that $\pi_2(v, b)$ never yields a negative result. The only possibility for negative values is that the second case of the definition is used i.e., $b < \pi_{\mathrm{cons}}(v)$ and that $\mathrm{cons}(P) < b$ holds. This is not possible, since $\pi_{\mathrm{cons}}(v)$ is a lower bound for $\mathrm{cons}(P)$.

A drawback of the new potential is, that it uses the value $\mathrm{cf}_{\max}$ as an upper bound for the charging rate. So the potential is inaccurate if the actual charging rate available on a path from s to t is much lower than $\mathrm{cf}_{\max}$. This can be the case if only normal charging stations are available but $\mathrm{cf}_{\max}$ is influenced by the existence of battery swapping stations somewhere else in the graph.

5.1.3 Incorporating Multiple Charging Station Types

Our previous potential functions assume, that a constant charging rate $\mathrm{cf}_{\max}$ is available everywhere in the graph. The potential is still admissible if this assumption does not hold but it may result in an inaccurate lower bound for the actual travel time. In order to improve the potential function we will now discuss how we can calculate a potential which takes into account reachability of charging stations.

The actually available charging rate for a vertex v depends on the set of charging stations directly reachable from v. Not all charging stations are directly reachable from v due to the fact that the reachability of electric vehicles is limited by the battery capacity. This means that we cannot drop all battery constraints if we want that the charging rate depends on the neighborhood of a vertex v.

As before we want to obtain a lower bound on the total travel time by performing a backwards search from t. If we compute Pareto-sets, similar to our approach for the forwards search, we would obtain the exact travel time, and therefore, a somehow perfect potential. This is of course not practicable since the computation of the potential would take as long as the algorithm without optimization. We will now analyze how such a backwards search works in detail and how it can be simplified, so that we can find admissible and accurate potential functions in short time.

Backwards Search With Travel-Time-Functions. We want to perform a backwards search originating from t similar to our forwards search. Since this search should take battery constraints and charging stations into account, again we will use the concept of SoC-functions. During forwards search these functions are used to map travel time onto the maximal available SoC. When performing a backwards search, our point of view is slightly different. Here we use travel-time-functions $\mathrm{tt}\colon \mathrm{B} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ which map available SoC onto the minimal travel time required to reach the target with the given SoC. So travel-time-functions can be seen as the inverse of SoC-functions. We use again piecewise linear functions to represent these functions, just as we did for the SoC-functions.

While being conceptually equivalent, the travel-time-functions used for backwards searches have some subtle differences compared to the SoC-functions used in the forwards search. The main difference is, that smaller values for the minimal required travel time are considered better. Because of that we have to adjust the definition of dominance. Given two travel-time-functions tt_A and tt_B, we define that $\mathrm{tt}_A \propto \mathrm{tt}_B$ holds only if-$\mathrm{tt}_A(b) \leq \mathrm{tt}_B(b)$ holds for all $b \in \mathrm{B}$. For the same reason ∞ is used as special value indicating that the path is not feasible, instead of $-\infty$. Furthermore, travel-time-functions are always convex. All these differences exist because a travel-time-function can be obtained from a SoC-function by swapping the time- and the SoC-axis.

Using this we can develop a backwards search starting at t with an initial travel-time-function which states that t is reachable from t in zero time and without requiring any energy, i.e., the travel-time-function defined by the supporting point $[(0,0)]$. Whenever a travel-time-function belonging to the vertex u is settled, all incoming edges of u are relaxed. Relaxing an edge e increases the SoC of the label by $\mathrm{cons}(e)$ and increases the minimal required travel time by $\mathrm{dt}(e)$. Thus $(\mathrm{cons}(e), \mathrm{dt}(e))$ is added to every supporting point of the travel-time-function. If the minimal SoC b, for which $\mathrm{tt}(b) < \infty$ holds, exceeds the battery capacity M, then the label can be discarded. Settling a vertex which also is a charging station works similar to the forwards search.

The backwards search stops as soon as all labels are settled. At this point we have for every vertex v a Pareto-set $\{\mathrm{tt}_1, \ldots, \mathrm{tt}_k\}$ of travel-time-functions, which map available SoC b onto the minimal Travel time required to reach t from v with an initial SoC of b. This Pareto-set can be used to define a potential function $\pi(v, b)$ as $\pi(v, b) := \min\{\mathrm{tt}_i(b) \mid 0 \leq i \leq k\}$. In order to find this lower bound, we simply search through all travel-time-functions in the Pareto-set and look for the one, which yields the lowest travel time for the given SoC b.

The whole procedure described so far is basically our default algorithm adapted to work as a backwards search. Using this as an actual potential would not provide any speedup, since computing the potential takes as long as an query without optimization. But now that we have an accurate potential we can use it as a starting point for further optimization. A main reason for Pareto searches being much slower than single criterion Dijkstra, is that the Pareto-sets can get quite large. This in turn increases the number of labels that get settled and edges which get relaxed.

Therefore, our main approach for a fast and accurate potential function is, to reduce the complexity of the Pareto-sets. We start by treating the Pareto-set as a whole, instead of the single entries. This means that we are transitioning from a label-setting algorithm to a label-correcting one.

Lower Bound Functions. In order to reduce the complexity of the Pareto-sets we want to view them as a whole, instead of as a set of many travel-time-functions. The easiest way to do so is replacing the Pareto-set with one single function which dominates all the travel-time-functions contained in the Pareto-set.

Consider a Pareto-set $\{\mathrm{tt}_1, \ldots, \mathrm{tt}_k\}$ of travel-time-functions. Given such a Pareto-set we define a new function $\mathrm{tt}\colon M \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ which, again, maps SoC onto minimal required travel time. The value of this function is defined as

$$\mathrm{tt}(b) := \min_{1 \leq i \leq k} \mathrm{tt}_i(b).$$

This function obviously dominates all travel-time-functions contained in the Pareto-set, while itself is not necessarily a travel-time-function, in the sense that it is in general neither continuous nor convex.

Now we can use any type of function as a lower bound for $\mathrm{tt}(b)$ in order to simplify the Pareto-set. We use such functions to replace all Pareto-sets in the backwards search, which results in a label correcting search algorithm. In order to be applicable in such a search, lower bound functions need to support some operations. First of all, there has to be a way of computing initial lower bound functions from a given Pareto-set. Furthermore, the lower bound functions need to support *linking* and *merging*, which each take two lower bound functions and map them to a new lower bound function.

Linking. Linking in general describes the result of consecutive events. This is the case if either two paths or edges are traversed successively, or a path or an edge is traversed after using a charging station.

We first consider the case of linking two paths or edges. Let therefore P and Q be two paths such that the last vertex of P is equal to the first vertex of Q. Note that in order to represent edges, a path containing only the two vertices of the edge, can be used. Furthermore let $\mathrm{tt}_P(b)$ and $\mathrm{tt}_Q(b)$ be two lower bound functions which map SoC b onto a lower bound for the travel time required to traverse P or Q. The result of linking these two functions is then defined to be a lower bound $\mathrm{tt}(b)$ for the conjunction of both paths, which we denote by $\mathrm{tt}_P \circ \mathrm{tt}_Q$. The result of linking two lower bound functions has to fulfill

$$(\mathrm{tt}_P \circ \mathrm{tt}_Q)(b) :\leq \min_{0 \leq b_A \leq b} \left(\mathrm{tt}_P(b_A) + \mathrm{tt}_Q(b - b_A)\right).$$

This definition reflects that both paths have to be traversed and that if we spend b_A energy on path P, we can only use the remaining energy $b - b_A$ to traverse path Q.

We now define how charging functions and paths or edges can be linked. Let therefore v be a charging station with its charging function cf_v. Furthermore let P be a path starting at v and let $\mathrm{tt}_P(b)$ be a lower bound function for the travel time needed to traverse P for an initial SoC of b. The result of linking the two functions cf_v and tt_P is denoted by $\mathrm{cf}_v \circ \mathrm{tt}_P$ and describes the minimal travel time needed to traverse P if we start with an initial SoC of b at v and are allowed to use the charging station before departing. The result of linking these two functions is defined as

$$(\mathrm{cf} \circ \mathrm{tt}_P)(b) :\leq \min_{\mathrm{ct} \geq 0} \left(\mathrm{tt}_P(\mathrm{cf}(b, \mathrm{ct})) + \mathrm{ct}\right)$$

This definition states simply that, before traversing the path P, an arbitrary amount of time can be spent for recharging. This of course increases the travel time but also the available energy for traversing P.

Merging. Merging is used when there are multiple paths between two vertices. In this case the merged lower bound function for these paths is defined as the minimal travel time required to traverse any of these paths for a given initial SoC b. Let P and Q be two u-v-paths and $\mathrm{tt}_P(b), \mathrm{tt}_Q(b)$ their corresponding lower bound functions. The result $\mathrm{tt}(b)$ of merging these two functions is a function which maps initial SoC b onto a lower bound for the travel time required to get from u to v, if either of the paths P and Q can be used. The merged function is denoted by $\mathrm{tt}_P \cup \mathrm{tt}_Q$ and has to fulfill

$$(\mathrm{tt}_P \cup \mathrm{tt}_Q)(b) :\leq \min\left(\mathrm{tt}_P(b),\ \mathrm{tt}_Q(b)\right).$$

Using lower bound functions which support these operations as labels we can perform a backwards and label correction search starting at t. As initial label for the vertex t, the constant function equal to 0 is used. As key for the lower bound function $\mathrm{tt}(b)$, which is used to sort them in the queue, we use $\mathrm{tt}(M)$, which is the minimal travel time needed to reach the target for any initial b. When settling a vertex, we first check if the vertex is a charging station. If this is the case, the charging function is linked with the current label. Afterwards, all incoming edges are relaxed. When relaxing an edge $e = (u, v)$, the current lower bound for u is linked with the lower bound for e, in order to obtain a new lower bound for v. If v had previously been visited, its current lower bound is merged with the new lower bound created during the edge relaxation.

5.1.4 Convex Lower Bound

As first approach for efficient lower bound functions we use again piecewise linear functions. Since the ideas proposed in the previous section emerged from piecewise linear travel-time-functions all needed operations can be implemented for this type of functions quite easily. Using the definitions of the previous section, our piecewise linear lower bound functions would contain one supporting point for every Pareto optimal solution. Thus we have to use a weaker lower bound in order to reduce the complexity.

A simple way of reducing the complexity of a piecewise linear function is to remove some of its supporting points. As we want to preserve a correct lower bound function, we cannot

Algorithm 5.1: Linking for Convex Lower Bound Functions

```
Input: Two sequences $A = [(b,\mathrm{tt})^A_1, \dots, (b,\mathrm{tt})^A_k], B = [(b,\mathrm{tt})^B_1, \dots, (b,\mathrm{tt})^B_\ell]$
Output: A sequence $C$ of supporting points representing $\mathrm{tt}_A \circ \mathrm{tt}_B$
1 $C \leftarrow \emptyset$
2 $i_A,\ i_B \leftarrow 0$
3 while $i_A \leq k$ and $i_B \leq \ell$ do
4     $C \leftarrow \mathrm{append}(C,\ A[i_A] + B[i_B])$
5     $m \leftarrow \min(\mathrm{slope}(A, i_A), \mathrm{slope}(B, i_B))$
6     if $\mathrm{slope}(A, i_A) \leq m$ then $i_A$++
7     if $\mathrm{slope}(B, i_B) \leq m$ then $i_B$++
```

remove arbitrary supporting points. When removing supporting points we have to ensure that the resulting function dominates the original one, i.e. itself is a lower bound for the original lower bound function. This means that we cannot remove those supporting points which are part of the convex hull of all supporting points.

Based on this observation our approach is to use convex and piecewise linear functions as lower bounds. We use a sequence $[(b_1, \mathrm{tt}_1), \dots, (b_k, \mathrm{tt}_k)]$ sorted by SoC to represent such a function. Next we have to show how these functions can be linked and merged. The result of these operations, as defined in the previous section, is not necessarily convex. Thus we can only use a convex lower bound for the actual result of these operations. Every time two functions are linked or merged, we use Grahams Scan [Gra72] to compute the convex hull of the resulting supporting points and discard all supporting points which are not part of the convex hull. Since our supporting points are already sorted, this can be done in a single linear sweep over the supporting points. Replacing the result of linking or merging with its convex lower bound will introduce an additional error, but it will hopefully also keep the number of needed supporting points low.

Linking. Linking two convex and piecewise linear functions can be done in linear time using a single sweep over the supporting points of both functions as shown in Algorithm 5.1. Given two sequences $A = [(b,\mathrm{tt})^A_1, \dots, (b,\mathrm{tt})^A_k]$ and $B = [(b,\mathrm{tt})^B_1, \dots, (b,\mathrm{tt})^B_\ell]$, which define the functions $\mathrm{tt}_A(b)$ and $\mathrm{tt}_B(b)$, the algorithm computes the supporting points of the function $(\mathrm{tt}_A \circ \mathrm{tt}_B)(b)$. Every supporting point of the resulting function is the sum of one supporting point from sequence A and one point from B. The first supporting point is the sum of the first points from both sequences A and B. The following supporting points are computed by adding the next supporting point of the currently steeper function with the same point as before of the other function.

We now show that the result of this procedure is the linked function, which is defined by

$$(\mathrm{tt}_A \circ \mathrm{tt}_B)\,(b) := \min_{0 \leq b_A \leq b} (\mathrm{tt}_A(b_A) + \mathrm{tt}_B(b - b_A)).$$

For $b < b^A_1 + b^B_1$ the linked function is equal to ∞, since there exists no b_A such that $\mathrm{tt}_A(b_A)$ and $\mathrm{tt}_B(b - b_A)$ are both less than ∞. For $b = b^A_1 + b^B_1$ there exists only one value for b_A such that the result is less than ∞. Thus we have $(\mathrm{tt}_A \circ \mathrm{tt}_B)\,(b^A_1 + b^B_1) = \mathrm{tt}^A_1 + \mathrm{tt}^B_1$, which corresponds to the first supporting point computed by Algorithm 5.1. We now assume that the first segment of the function tt_A is steeper than the first segment of function tt_B. In this case our algorithm uses $(b^A_2 + b^B_1, \mathrm{tt}^A_2 + \mathrm{tt}^B_1)$ as second supporting point of the linked function. This is equivalent to the definition of linking, since there exists no b_A such that

$$(\mathrm{tt}_A \circ \mathrm{tt}_B)\,(b^A_2 + b^B_1) = \mathrm{tt}_A(b_A) + \mathrm{tt}_B(b^A_2 + b^B_1 - b_A)$$

is less than $\mathrm{tt}^A_2 + \mathrm{tt}^B_1$. For $b_A = b^A_2$ we get $\mathrm{tt}^A_2 + \mathrm{tt}^B_1$ as travel time, which is the value of the computed supporting point. Choosing $b_A > b^A_2$ leads to a travel time of ∞,

Algorithm 5.2: Merging for Convex Lower Bound Functions

```
    Input: Two sequences $A = [(b, \mathrm{tt})^A_1, \dots, (b, \mathrm{tt})^A_k], B = [(b, \mathrm{tt})^B_1, \dots, (b, \mathrm{tt})^B_\ell]$
    Output: A sequence $C$ of supporting points representing $\mathrm{tt}_A \cup \mathrm{tt}_B$
1  $C \leftarrow \emptyset$
2  $i_A,\ i_B \leftarrow 0$
3  while $i_A \leq k$ and $i_B \leq \ell$ do
4      $\mathrm{tt} \leftarrow \min(A[i_A].\mathrm{tt}, B[i_B].\mathrm{tt})$
5      if $A[i_A].\mathrm{tt} \leq \mathrm{tt}$ then $C \leftarrow \mathrm{append}(C,\ A[i_A{+}{+}])$
6      if $B[i_B].\mathrm{tt} \leq \mathrm{tt}$ then $C \leftarrow \mathrm{append}(C,\ B[i_B{+}{+}])$
7  while $i_A \leq k$ do
8      $C \leftarrow \mathrm{append}(C,\ A[i_A{+}{+}])$
9  while $i_B \leq \ell$ do
10     $C \leftarrow \mathrm{append}(C,\ B[i_B{+}{+}])$
```

because $b^A_2 + b^B_1 - b_A$ is less than the minimal SoC required to traverse B. Finally, choosing $b_A < b^A_2$ cannot yield a travel time less than $\mathrm{tt}^A_2 + \mathrm{tt}^B_1$, because of the slopes of both functions. A smaller value for b_A means that the SoC available for traversing A decreases while the SoC for B increases. This means traversing A will take longer while traversing B will take less time. But, since the slope of A is steeper, the additional travel time needed for A will be greater than the travel time saved on B. Thus the overall travel time will increase if b_A is less than b^A_2. We can repeat the same argument for the remaining supporting points of the linked function. Therefore, the function computed by our algorithm is indeed the linked function.

Merging. Merging two convex and piecewise linear functions is even easier than linking them. In order to ensure that

$$(\mathrm{tt}_P \cup \mathrm{tt}_Q)(b) \leq \min(\mathrm{tt}_P(b),\ \mathrm{tt}_Q(b))$$

holds we simply have to merge the supporting points of both functions into one sequence which is again sorted by SoC. Since simply merging the supporting points of both lower bound functions yields in general a function which is not convex, we use Grahams Scan simultaneously in order to ensure that the resulting sequence of supporting points is convex. An exemplary pseudo code implementation of the merge operation is provided in Algorithm 5.2.

Since we have defined all required operations, we now can perform a label correcting search starting at the target vertex which uses piecewise linear and convex functions as label. Doing so results in one lower bound function $\mathrm{tt}_v \colon \mathrm{B} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ for every vertex v. Note that if the backwards search did not reach a vertex v, which means that there exists no feasible v-t-P, then the lower bound function for this vertex is constant and equal to ∞. This lower bound function $travelTime_v$ can then be used directly as a potential for the A* search.

$$\pi_3(v, b) := \mathrm{tt}_v(b).$$

5.1.5 Parametric Lower Bound

Another approach is to use a parametric equation as lower bound for the Pareto-sets. The big advantage of such a solution is, that the complexity of a parametric equation is constant. This means that the lower bound function only needs constant memory space, independent of the original Pareto-sets complexity. We propose to use hyperbolic functions of the form

$$y = \frac{\alpha}{x - \beta} + \gamma$$

as lower bound for the minimal required travel time depending on the initial SoC. Furthermore, we use explicit values mintt and min$b$ which are lower bounds for the travel time and the SoC required to traverse a path. Incorporating these values yields lower bound functions $\mathrm{tt}(b)$ of the form

$$\mathrm{tt}(b) := \begin{cases} \infty & \text{if } b < \min b \\ \max(\frac{\alpha}{b-\beta} + \gamma,\ \mathrm{mintt}) & \text{else,} \end{cases}$$

which are represented by quintuples $(\alpha, \beta, \gamma, \mathrm{mintt}, \min b)$. Since not every combination of these five values yields a meaningful lower bound function we introduce some additional restrictions for the values of α, β, and γ, which ensure that the resulting function is convex:

$$\begin{aligned} \alpha &\geq 0 \\ \beta &\leq \min b \\ \gamma &\leq \mathrm{mintt}\,. \end{aligned}$$

We can now try to use these functions in the same way as we have used the piecewise linear and convex functions before. First we have to consider how we can compute the five values from a given Pareto-set of paths or from a single edge. In the case of piecewise linear and convex functions we used Grahams Scan to compute the convex hull of the given Pareto points. Computing an initial lower bound is slightly more complicated for our new hyperbolic functions.

Given a Pareto-set $[(b_1, \mathrm{tt}_1), (b_2, \mathrm{tt}_2), (b_3, \mathrm{tt}_3)]$ consisting of exactly three points, there exists a unique hyperbolic function, such that all three points are contained in the graph of this function. First of all we can compute the minimal values for required travel time and SoC

$$\begin{aligned} \mathrm{mintt} &:= \min\{\mathrm{tt}_1, \mathrm{tt}_2, \mathrm{tt}_3\} \\ \min b &:= \min\{b_1, b_2, b_3\}. \end{aligned}$$

Furthermore, we know that all three points are contained in the graph of a hyperbolic function, which leads to a system of three equations with three unknown variables. We solve this system for α, β, and γ and obtain:

$$\begin{aligned} \beta &:= \frac{(\mathrm{tt}_2 - \mathrm{tt}_1)b_1b_2 + (\mathrm{tt}_1 - \mathrm{tt}_3)b_1b_3 + (\mathrm{tt}_3 - \mathrm{tt}_2)b_2b_3}{(\mathrm{tt}_1 - \mathrm{tt}_2)b_3 + (\mathrm{tt}_3 - \mathrm{tt}_1)b_2 + (\mathrm{tt}_2 - \mathrm{tt}_3)b_1} \\ \gamma &:= \frac{b_1\,\mathrm{tt}_1 - b_2\,\mathrm{tt}_2 - (\mathrm{tt}_1 - \mathrm{tt}_2)\beta}{b_1 - b_2} \\ \alpha &:= (\mathrm{tt}_1 - \gamma)(b_1 - \beta). \end{aligned}$$

If less than three points are given, we can use degenerated forms of hyperbolic functions. In the case of two Pareto points we generate a third point approaching the arithmetic mean of the two given points. This gives us

$$\beta = \frac{b_1 + b_2 - b_1b_2}{\varepsilon} \quad \text{for} \quad \varepsilon > 0$$

as value for β. Afterwards, we use the definition for γ and α from above. In the case that only one point (b, tt) is given, we set $\alpha = 0$, $\beta = b$ and $\gamma = \mathrm{tt}$. Examples for all these cases are provided in Figure 5.1.

In the case that more than three Pareto points are given, there exists no simple closed formula for a good hyperbolic lower bound. Thus we have to search for a hyperbolic lower bound function which is close to all given Pareto points. The best hyperbolic function we

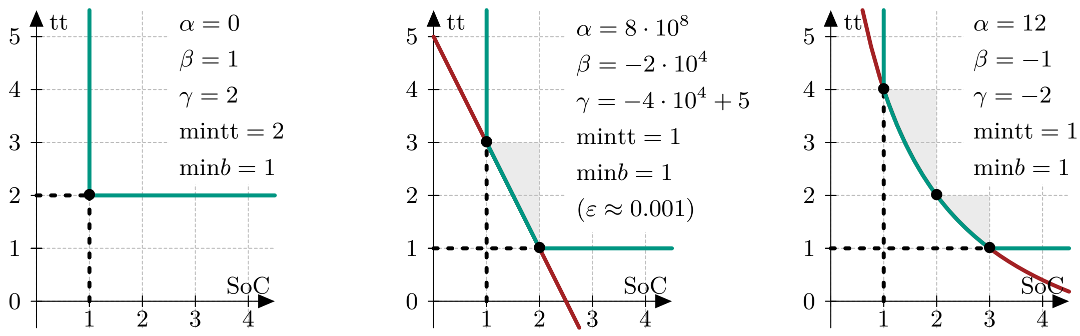

Figure 5.1: Three examples for hyperbolic lower bound functions in green. Left: A lower bound for a single Pareto point $(1, 2)$. Middle: A lower bound for two Pareto points $(1, 3), (2, 1)$. Right: A lower bound for three Pareto points $(1, 4), (2, 2), (3, 1)$. The red function shows the hyperbolic function without the limits mintt and min$b$. The shaded area depicts the deviation from the original Pareto-set.

could find, is one that is a lower bound for all Pareto points and minimizes the deviation from the original Pareto-set. As deviation we define the area enclosed by the hyperbolic function and the Pareto front.

If we additionally demand that at least three Pareto points are contained in the hyperbolic function, then we can simply search through all functions defined by a subset of three points from the Pareto-set. For every such subset we compute the uniquely defined hyperbolic lower bound function. We then check if this function is also a lower bound for the remaining points in the Pareto-set. If this is the case, we compute the actual deviation from the Pareto front. Finally, we keep only the hyperbolic lower bound function with the minimal deviation.

This search is not really efficient, but computing initial lower bound functions has to be done only once and can even be part of preprocessing. Furthermore, there is normally only one Pareto per edge. Therefore, we can often compute initial lower bounds efficient using the closed formula. Next we show how hyperbolic lower bounds can be linked.

Linking. Recall, that linking for arbitrary lower bound functions $\mathrm{tt}_A, \mathrm{tt}_B$ is defined as

$$\mathrm{tt}(b) = (\mathrm{tt}_A \circ \mathrm{tt}_B)(b) := \min_{0 \le b_A \le b} (\mathrm{tt}_A(b_A) + \mathrm{tt}_B(b - b_A)) .$$

Together with the definition of the hyperbolic lower bound functions we can see that the lower bounds $\min \mathrm{tt}$ and $\min b$ are simply the sum of the corresponding values from the function tt_A and tt_B. This means $\mathrm{mintt} := \mathrm{mintt}_A + \mathrm{mintt}_B$ and $\mathrm{min}b := \mathrm{min}b_A + \mathrm{min}b_B$.

Now we look at the function $f(b_A) := (\mathrm{tt}_A(b_A) + \mathrm{tt}_B(b - b_A))$ for a fixed value of b. For tt_A and tt_B we insert the corresponding hyperbolic functions without using their explicit lower bounds (mintt and min$b$). Thus we get

$$f(b_A) := \frac{\alpha_A}{b_A - \beta_A} + \gamma_A + \frac{\alpha_B}{b - b_A - \beta_B} + \gamma_B .$$

Since ignoring the bounds mintt and min$b$ of our hyperbolic functions only decreases their value, we can use the function f to calculate a lower bound for the linked function $\mathrm{tt}(b)$

$$\min_{0 \le b_A \le b} f(b_A) \;\le\; (\mathrm{tt}_A \circ \mathrm{tt}_B)(b) \;=\; \mathrm{tt}(b).$$

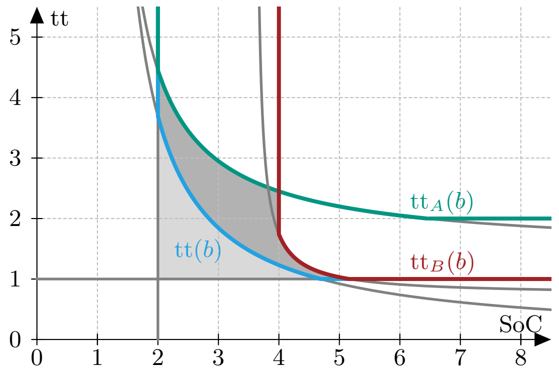

Figure 5.2: Example of two hyperbolic functions ($\mathrm{tt}_A(b)$ and $\mathrm{tt}_B(b)$), and how the result of merging them could look like ($\mathrm{tt}(b)$). The area shaded in dark grey corresponds to the error of the merge operation. Minimizing the area shaded in dark grey (the error) will maximize the area shaded in light grey.

Next we determine the minimal value of the function f. We do so by differentiating f and analyze where it is equal to 0, which leads to

$$f'(b'_A) \;=\; \frac{-\alpha_A}{(b'_A-\beta_A)^2}+\frac{\alpha_B}{(b-b'_A-\beta_B)^2} \;\overset{!}{=}\; 0.$$

Solving this equation for b'_A yields

$$b'_A \;=\; \frac{\alpha_A(b-\beta_B)-\alpha_B\beta_A-\sqrt{\alpha_A\alpha_B}(b-\beta_A-\beta_B)}{\alpha_A-\alpha_B}.$$

Using this value b'_A as argument of the function f yields

$$\min_{0\le b_A\le b} f(b_A) \;=\; f(b'_A) \;=\; \frac{\alpha_A+\alpha_B+2\sqrt{\alpha_A\alpha_B}}{b-(\beta_A+\beta_B)}+(\gamma_A+\gamma_B),$$

which is a hyperbolic function of the argument b. Thus we can use it as the linked function $\mathrm{tt}(b)$, which therefore is defined by the three values

$$\begin{aligned}\alpha &= \alpha_A+\alpha_B+2\sqrt{\alpha_A\alpha_B}\\ \beta &= \beta_A+\beta_B\\ \gamma &= \gamma_A+\gamma_B.\end{aligned}$$

Merging. The last operation we need to define for hyperbolic functions, so that we can use them as lower bound functions, is merging. Unlike the other operations, merging two hyperbolic functions causes some problems. The main reason for this is that the minimum of two hyperbolic functions is in general not a hyperbolic function. An example for this is shown in Figure 5.2. Thus, if we want to merge two hyperbolic functions, we can only try to find a lower bound for the exact minimum of the two functions.

So we are searching for a hyperbolic function $\mathrm{tt}(b)$, which satisfies the equation

$$\mathrm{tt}(b) := (\mathrm{tt}_P \cup \mathrm{tt}_Q)(b) \le \min(\mathrm{tt}_P(b),\ \mathrm{tt}_Q(b)),$$

and is as close as possible to the two functions $\mathrm{tt}_A(b)$ and $\mathrm{tt}_B(b)$. We can quantify the deviation from the optimal solution of this problem, by the area enclosed by the three functions $\mathrm{tt}(b)$, $\mathrm{tt}_A(b)$ and $\mathrm{tt}_B(b)$. This area is shaded in dark gray in Figure 5.2. So we want that $\mathrm{tt}(b)$ minimizes this area.

Now we have to find a quintuple $(\alpha, \beta, \gamma, \text{mintt}, \text{min}b)$, which fulfills the conditions mentioned above. It is easy to see that mintt and min$b$ have to be defined as

$$\begin{aligned} \text{mintt} &:= \min\left(\text{mintt}_A, \text{mintt}_B\right) \\ \text{min}b &:= \min\left(\text{min}b_A, \text{min}b_B\right). \end{aligned}$$

This helps us to simplify the objective function of our optimization problem. Instead of minimizing the area enclosed by the three functions $\text{tt}(b)$, $\text{tt}_A(b)$ and $\text{tt}_B(b)$, we can now maximize the area beneath $\text{tt}(b)$, which is additionally bounded by mintt and min$b$. This area is shaded light grey in Figure 5.2. Since this area is only bounded by one hyperbolic function, it is much easier to compute. The actual value of this area $\mathcal{A}$ is given by

$$\begin{aligned} \mathcal{A}(\alpha, \beta, \gamma) &:= \int\limits_{\text{min}b}^{\frac{\alpha}{\text{mintt}-\gamma}+\beta} \left(\frac{\alpha}{b-\beta} + \gamma - \text{mintt}\right) \mathrm{d}b \\ &= \alpha \log\left(\frac{\alpha}{(\text{min}b - \beta)(\text{mintt} - \gamma)}\right) + (\text{min}b - \beta)(\text{mintt} - \gamma) - \alpha. \end{aligned}$$

Now we can apply some numerical optimization technique in order to maximize the function $\mathcal{A}(\alpha, \beta, \gamma)$ with the side conditions that $\text{tt}(b) \le \text{tt}_A(b)$ and $\text{tt}(b) \le \text{tt}_B(b)$ hold for every $b \in \mathrm{B}$. Unfortunately this is relatively slow compared with the constant running time need for linking hyperbolic functions or the linear running times when using piecewise linear and convex functions.

5.2 Contraction Hierarchies

The second speedup technique we adapt for our scenario are Contraction Hierarchies (CH). The CH algorithm was first introduced by Geisberger et al. [GSSD08], where it was used to speedup a single criterion shortest path search. The approach of Storandt [Sto12b] already adapts CH for electric vehicle routing with charging stations, although they only consider battery swapping stations. For building the CH they use the techniques, which they introduced in [Sto12c].

An essential component of the algorithm proposed in [Sto12b] is, that the vertices which represent charging stations are not contracted during the CH preprocessing. Thus the impact of charging stations only has to be considered in the query algorithm but not during preprocessing. We also use this approach for our CH preprocessing. Apart from that, crucial ingredients for the CH preprocessing are the vertex ordering, in which the vertices get contracted, and the witness search, which is used to determine if a shortcut es needed or not.

5.2.1 Heuristic Vertex Ordering

In their work, Geisberger et al. introduce and evaluate several criteria, which can be used to determine the next vertex to be contracted. In our implementation we use three of the introduced criteria, namely *edge difference*, *contracted neighbors* and *search space depth*.

Edge Difference. The edge difference vor a certain vertex v describes the relation between removed edges and newly added shortcuts when contracting the vertex v. In the original work of Geisberger et al. this value was defined as the difference between the number of shortcuts that have to be added when contracting v and the degree of v, which is the number of edges that get removed when contracting v. The number of shortcuts that have to be added is, for this purpose, determined by simulating the contraction of v. The purpose of this criterion is, to keep the average degree of the uncontracted small.

In our implementation make some small adjustments to this criterion. First of all, the original CH preprocessing was designed for the single criterion shortest path problem. Therefore there exits a unique shortest distance between any pair of two vertices, leading to at most one shortcut between two vertices. As shown before, when considering our EV routing problem, we have to deal with Pareto-sets of optimal paths between pairs of vertices. So one shortcut can represent an arbitrary number of paths, depending on the size of the corresponding Pareto-set. Because of this we do not use the number of added shortcuts and the degree of a vertex v, but the sum of the sizes of the Pareto-sets associated with these edges.

Furthermore we use the quotient of these two values instead of their absolute difference. Thus our final value for the vertex v using this criterion is given by

$$\frac{\sum\limits_{e \in \mathrm{SC}(v)} \mathrm{ParetoSetSize}(e)}{\sum\limits_{(v,u) \in E} \mathrm{ParetoSetSize}(v,u) + \sum\limits_{(u,v) \in E} \mathrm{ParetoSetSize}(u,v)},$$

where $\mathrm{SC}(v)$ denotes the set of shortcuts, which have to be added when contracting v, and $\mathrm{ParetoSetSize}(e)$ denotes the size of the Pareto set associated with the edge ore shortcut e. All edges in the input graph have of course one fixed driving time and energy consumption, thus their Pareto Set size is counted as 1. Shortcuts, on the other hand, may contain Pareto sets with multiple entries, whenever the exit multiple non dominating paths between the head and the tail of the shortcut, as we have seen in Section 2.3.4.

Contracted Neighbors. Since the main aim of the CH preprocessing is, to speedup query time, it is desirable that the search space of the later query algorithm is minimized. The contracted neighbors criterion tries to achieve this by preventing that many vertices which are neighbors of each other get contracted successively. The criterion simply counts the number of neighbors of a vertex, which already got contracted. This number is initially zero for every vertex. Every time a vertex v gets contracted, the number is incremented by one for all neighbors of v.

Search Space Depth. Another approach for minimizing the later search space size is, to minimize the depth of the search tree. The search space depth is a rough approximation for the depth of the vertex v in a search tree during some query. This criterion again is initially zero for all vertices. When contracting a vertex v, the depth off all its neighbors may increase by one. Therefore the new depth of its neighbors is set to the depth of v plus one, if it was not already higher.

A sum of all three criteria is used to evaluate how important a vertex is. A higher value implies that the vertex is more important, thus the vertex with the smallest sum of the three criteria gets contracted next. As we will see in the experimental section, the Pareto-sets as well as the average vertex degree tends to get very high during the preprocessing. We try to compensate for that by multiplying the edge difference criterion with the factor 16 when computing the sum of the three criteria.

5.2.2 Witness Search

Another essential component of the CH preprocessing is the witness search. As already discussed in the preliminaries section, we have to use a Pareto search as witness search, if we do not want to add unnecessary shortcuts. Unfortunately such a search is relative slow compared to a single criterion search. This difference becomes particularly obvious, if the core graph already features a high average vertex degree and if the shortcuts contain already large Pareto-sets.

In order to make the preprocessing phase a bit faster, we allow that some unnecessary shortcuts are added, if this reduces the time spent for the witness search. So we do not need to perform a full and correct Pareto search, but can use a heuristic algorithm instead. We only have to make sure that the heuristic search considers only feasible paths, i.e., the computed travel time is never shorter than the correct travel time, which would be computed by a full Pareto search.

Our first improvement is to limit the size of the Pareto-sets used as labels during the search. We use a fixed upper bound for all involved Pareto-sets. If, due to an edge relaxation, the Pareto-set of some vertex exceeds this upper bound we simply delete some of the entries, so that the size is again lower than the given bound. When deleting entries from a Pareto-set we try to maintain the overall shape, so that the resulting set is a good approximation for the initial set. We do so by comparing the entries of the Pareto-set. We search for the two points contained in the set with the minimal euclidean distance and delete one of them. In our implementation we use 10 as an upper bound for the Pareto-set size.

Another possibility to reduce the computation time is to limit the depth of the search tree. For this purpose we extend each label by a counter, counting the number of edges, that have been relaxed in order to create the label. When relaxing an edge, the depth of the newly created label is given by the depth of the current label plus one. If the depth of a label exceeds the given limit, the label is simply ignored. In our implementation we use a maximal search depth of 15.

The witness search is not only used when contracting a node, but also when the contraction is simulated in order to compute the edge difference criterion. So the witness search is actually performed twice for every vertex. This can, of course, be optimized. When computing the edge difference, we do not only count the size of the Pareto-set of the shortcuts, but we also save all shortcuts that have to be added if the vertex gets contracted. Thus we do not need to repeat the witness search if the vertex gets actually contracted.

5.2.3 Query Algorithm and SoC-Functions

Our charging station propagating algorithm from Chapter 4.2 only needs minor modifications, so that it can be used with a CH instead of a graph without preprocessing.

Just as the normal CH query algorithm introduced by Geisberger et al., our algorithm relaxes only edges or shortcuts in the direction of increasing vertex importance. Recall that a vertex u was defined to be more important than another vertex v, if u got contracted after v. As we do not allow that charging stations get contracted, our CH contains a core of uncontracted vertices. All uncontracted vertices are defined to be equally important, thus our algorithm has to relax all edges or shortcuts between vertices in the core graph.

Our search algorithm, using SoC-functions, is actually only meant to be used as a forwards search. A CH query, however, requires a bidirectional search. We solve this problem by splitting the search algorithm in two phases. First we perform a backwards search starting at t, until we reach the core graph. This search can be done without using SoC-functions, because all charging stations are contained in the core graph. We stop the backwards search as soon as the core is reached, thus we cannot encounter any charging stations and our backwards search needs only to link the energy consumption functions of the relaxed edges. Afterwards, we add additional shortcuts from the core vertices, which were reached by the backwards search, to t to our graph. Furthermore, we give t the highest priority so that these new shortcuts are relaxed by the later forwards search. After this is done we start a forwards search from s which now uses SoC-functions.

The last difference between our CH query algorithm and the plain search algorithm using SoC-functions is the relaxation of shortcut edges. A shortcut from u to v potentially represents a complete Pareto-set of paths from u to v. We relax each of these paths separately as if they were different edges.

Relaxing a complete path P instead of an edge is actually not a huge difference, if the SoC $b(\mathrm{tt})$, which is used as label, is represented by the quadruple $(v, b, \mathrm{c}, \mathrm{tt})$. The relaxed SoC-function, which is denoted by $(b \circ P)\,(\mathrm{tt})$, is in this case defined by the quadruple $(v, b, \mathrm{c} \circ \mathrm{c}(P), \mathrm{tt} + \mathrm{dt}(P))$. This definition reflects, that relaxing a path changes neither the last seen charging station nor the SoC b with which this charging station was reached. But relaxing a path increases the minimal travel time by the time needed to traverse the path. Furthermore, it changes the energy consumption on the path from the last charging station to the last vertex of P. We compensate for that by linking the current energy consumption function c with the energy consumption function $\mathrm{c}(P)$ of the path P.

5.3 Combining CH and A*

As the size of Pareto-sets can grow exponentially in the size of the graph, it is sometimes simply not possible to compute a complete CH. We can stop the CH preprocessing at any given time, for example if a certain time has passed, or if the average degree in the uncontracted graph gets too high. Stopping the preprocessing will leave us with a hopefully small core graph. We now can use the A* speedup technique restricted to the core graph.

The only difference to using A* on the input graph is, that the edges in the core graph already contain Pareto-sets instead of a single travel time, consumption pair. During forwards and backwards search the Pareto edges are relaxed as if they were separate edges, leading to the same vertex. When using lower bound functions, in order to compute the potential function, these Pareto-sets can be approximated by lower bound functions before the actual backwards search starts. This can already be done during preprocessing.

If we cannot compute a complete CH, we perform a reordering on the vertices, such that all core vertices are stored within consecutive memory.

6. Heuristics

In this chapter, we introduce several heuristics for EVRC. Instead of insisting on finding the optimal solution, we now only aim for fast algorithms. We start by reviving the conventional car driver's approach which was first introduced in Chapter 4.1. Afterwards, we develop heuristics based on our algorithms from Chapters 4.2 and 5.

6.1 Conventional Car Driver's Approach (CCDA)

As we have shown before, it is not sufficient to treat charging stations for EVs in the same way as it is done with gas stations for conventional vehicles. While we were not able to develop an optimal algorithm using this approach it might still lead to a good heuristic.

The basic idea of the conventional car driver's approach is, that we restrict ourselves to using only shortest paths without considering battery constraints. This restriction should be fulfilled between any two stops, with a stop being defined as starting at s, arriving at t, or using a charging station. Doing so, the path between any two charging stations is uniquely defined. So, when computing a path from s to t, the algorithm only has to choose a sequence of charging stations.

Preprocessing. Since the path connecting any two charging stations is uniquely defined, it can be precomputed. We now introduce a preprocessing step where we compute a shortcut graph $G' = (V', E')$, similar to the auxiliary graph proposed in [SF12], containing all charging stations as vertices, i.e., $V' := \mathrm{CS}$. This graph contains an edge connecting two charging stations, if and only if the shortest path (regarding driving time) between these two charging stations is also a feasible path (regarding battery constraints) when departing with a fully charged battery.

For this purpose, we compute a shortest path tree $\mathcal{T}_u$ for every charging station $v \in \mathrm{CS}$. This shortest path tree can be computed using Dijkstra's algorithm. In addition to dist$[\cdot]$ and parent$[\cdot]$, we also keep track of the linked energy consumption function c for every vertex in $\mathcal{T}_u$. After we computed $\mathcal{T}_u$, we check for every charging station $v \in \mathrm{CS}$ whether the linked energy consumption function of the path from u to v is feasible. If this is the case, we add an edge $e = (u, v)$ to the shortcut graph G'. Since we only add edges for such vertices, where the consumption function is feasible, we may as well stop the execution of Dijkstra's algorithm as soon as the consumption functions for all vertices in Q are unfeasible. This does not violate correctness of the algorithm, because relaxing an vertex with an unfeasible battery state can never lead to a vertex with a feasible battery state.

Finally, we define an edge weight function on the computed shortcut graph G'. Recall that every edge $e = (u, v) \in E'$ represents a shortest path from u to v in G. We use the driving time of this path as driving time $\mathrm{dt}(e)$ for the edge e. Furthermore, we know the linked energy consumption function of this path. We use this function to describe the energy consumption of e. As before, we represent this function using three values, hence we get the three additional edge weights $\mathrm{cost}(e)$, $\mathrm{minIn}(e)$, and $\mathrm{maxOut}(e)$.

Query Algorithm. Our query algorithm gets G' as additional input and operates in two phases. The first phase uses Dijkstra's algorithm to calculate two shortest path trees, one in forwards direction rooted at s and one in backwards direction rooted at t. Just as during preprocessing, we keep track of the linked energy consumption functions and stop the computation as soon as the queue contains only vertices with a unfeasible energy consumption functions. If the forwards search reaches t and the initial SoC $b(s)$ is sufficient to reach t, we can stop the computation and output the found path. Otherwise, we compute the set CS_s containing all charging stations reached from s and CS_t containing all stations from which t is reachable. Afterwards, we proceed with the second phase of the algorithm.

We now know that t is not reachable from s without recharging. Moreover, we know all charging stations reachable from s and all charging stations from which t is reachable. Now we can use the shortcut graph G' to compute the shortest path between any pair of vertices in CS_s and CS_t. For this purpose, we temporarily add s and t to V' and connect them to the sets of reachable charging stations. This is done by adding edges $e = (s, u)$ for every $u \in \mathrm{CS}_s$ and $e' = (v, t)$ for every $v \in \mathrm{CS}_t$ to E'. The driving time and consumption values of these edges are the ones of the associated paths in G as they were computed in phase one. Afterwards, we use a slightly modified version of Dijkstra's algorithm on G' to compute the final path connecting s and t. We use an additional array $b[\cdot]$ to keep track of the current SoC for every vertex we visit. This means that we maintain a label (tt, b) consisting of travel time and SoC for every vertex. It is now fairly easy to decide how much time has to be spent for charging when visiting a charging station.

Every time an shortcut edge $e = (u, v) \in E'$ is relaxed, we know the current SoC $b[u]$ at u. Therefore, we can define the function $f(\mathrm{ct}) := \mathrm{cf}_u(b[u], \mathrm{ct})$ which maps charging time at the charging station u to the thereby reached SoC. Furthermore, we know the minimal SoC needed for traversing e given by $\mathrm{minIn}(e)$. Now we can recharge at u until the state of charge is exactly $\mathrm{minIn}(e)$. By doing so we can compute a new label (tt', b') for v. The travel time $\mathrm{tt}' := \mathrm{tt}[u] + f^{-1}(\mathrm{minIn}(e)) + \mathrm{dt}(e)$ is given as the sum of the time needed to reach u, the recharging time and the driving time for the edge e. The new SoC for the vertex v is given by $b' := \mathrm{minIn}(e)$. As before the edge e gets only relaxed if tt' is less than the current travel time $\mathrm{tt}[v]$ for reaching v.

Note that we only relax such edges $e = (u, v) \in E'$ where $b[u] < \mathrm{minIn}(e)$ holds. The reason for this is the precondition that we cannot reach v without recharging. Otherwise there would be an edge from the parent of u to v in G'.

Complexity. During preprocessing we compute one shortest path tree for every charging station, using basically plain Dijkstra. Therefore, the time complexity of the preprocessing phase is given by $\mathcal{O}(|\,\mathrm{CS}\,| \cdot (n + m) \log \cdot n)$. The preprocessing might add up to $|\,\mathrm{CS}\,|^2$ new edges, thus we get $m' \in \mathcal{O}(m + |\,\mathrm{CS}\,|^2)$.

The query algorithm consists of up to three invocations of Dijkstra's algorithm. Two invocations are plain Dijkstra, and compute the set of charging stations reachable from s and the set of charging stations from which t is reachable. The third invocation uses Dijkstra's algorithm with a slightly modified relax operation. But this modified edge relaxation can still be done in constant time. Therefore, the overall worst case complexity of the query algorithm is equivalent to the worst case complexity of Dijkstra's algorithm, which is $\mathcal{O}((n + m') \log \cdot n) = \mathcal{O}((n + m + |\,\mathrm{CS}\,|^2) \log \cdot n)$ when using a Binary Heap.

6.2 Optimal Travel Time for Given Charging Speed

In this section we add heuristics to our algorithm developed in Chapter 4 and 5. We no longer insist on finding the optimal solution, if this reduces the computation time. The starting point for this heuristic is our base algorithm with both speedup techniques, A* and CH, enabled.

Our first heuristic improvement focuses on edge relaxations. After the CH preprocessing, most edges in the core graph are shortcuts. Recall that a shortcut from u to v has a Pareto-set assigned to it, where each entry corresponds to a Pareto optimal path from u to v. When relaxing such a shortcut edge, each of the Pareto-set's entries has to be linked with the current label for vertex u, which generates a new label for the vertex v. Furthermore, for every new label, it has to be checked if it is dominated by another label of the vertex v, or if itself dominates some of these labels.

The Pareto-set of a shortcut edge, can potentially be very large. This leads to a large number of new labels being created, as well as a large number of dominance checks that have to be performed. On the other hand, if the optimal route contains the vertices u and v, it can at most use one of the paths represented by the Pareto-sets entries. Therefore, our approach is to guess which of the Pareto-sets entries is likely to be part of the optimal s-t-path. Then our algorithm considers only this single entry, when relaxing a shortcut edge.

Next, we describe the criterion we use to determine the single entry from the Pareto-set that gets relaxed. In order to find this entry, we have to compare the entries in the Pareto-set. When comparing two non dominating paths between the same vertices, we make the following general observations.

- It is not meaningful to use the faster path, which consumes more energy, if the lost energy has to be recharged afterwards and recharging takes longer than the time saved by using the faster path.
- It is not meaningful to use the slower path, in order to save energy, if the saved energy could also be acquired by using a charging station and recharging takes less time than the additional time needed to traverse the slower path.

Basically, this means that, under some circumstances, the optimal path is simply the one, which minimizes the sum of driving time and the time required to recharge the amount of energy used to traverse the path. This sum is actually equivalent to the ω edge weight function which we introduced in Chapter 5.1.2, provided that recharging is possible with a constant charging speed of $\mathrm{cf}_{\max}$. Recall that the ω edge weight function was defined as

$$\omega(e) := \mathrm{dt}(e) + \frac{\mathrm{cons}(e)}{\mathrm{cf}_{\max}}.$$

In fact, we can show that ω is optimal under certain conditions, which leads to the following theorem.

Theorem 6.1. *We are searching for the fastest feasible s-t-path. Given a Pareto-set $\mathcal{P}$ containing all non dominating paths from u to v, the path $P \in \mathcal{P}$ which minimizes $\omega(P)$, is a subpath of the fastest feasible s-t-path, provided that the following conditions are satisfied.*

- *The fastest path contains u and v in this order.*
- *The SoC at u is not sufficient to reach t without recharging, regardless of which path from $\mathcal{P}$ is used.*
- *There occurs no over-charging on the path from u to the next used charging station.*
- *No matter which path is used, there is always a charging station available before the battery depletes, and the charging speed is always $\mathrm{cf}_{\max}$.*

Proof. Let $P \in \mathcal{P}$ be the path which minimizes $\omega(P)$ and tt_P the total travel time required to reach t when using P. Now we assume, for contradiction, that all four conditions are satisfied and that there exists another path $P' \in \mathcal{P}$ with a total travel time $\mathrm{tt}_{P'} < \mathrm{tt}_P$ for reaching t.

Case 1. We first consider the case of $\mathrm{dt}(P') < \mathrm{dt}(P)$, which means that traversing P' is faster than traversing P. Since all paths in $\mathcal{P}$ are non dominating, it also means that the energy consumption of P' is higher than the one of P, i.e., $\mathrm{cons}(P') > \mathrm{cons}(P)$.

Consider the fastest s-t-path that has a total travel time of $\mathrm{tt}_{P'}$ and contains P' as a subpath. Next, we replace the occurrence of P' in this s-t-path by P. This increases the total travel time of the path by $\mathrm{dt}(P) - \mathrm{dt}(P')$. Moreover, the total path remains feasible since the energy consumption did not increase.

Because t is not reachable without recharging, regardless of whether we use P or P', there has to be a charging station on the s-t-path which we use. The third condition tells us now, that no over-charging occurred between u and this charging station. Therefore, when replacing P' by P, our SoC at this charging station increases by $\mathrm{cons}(P') - \mathrm{cons}(P)$. Thus, we do not need to recharge this amount of energy, which reduces the charging time by $(\mathrm{cons}(P') - \mathrm{cons}(P)) / \mathrm{cf}_{\max}$. Finally, the travel time for reaching t when replacing P' with P is given by

$$\mathrm{tt}_{P'} + \big(\mathrm{dt}(P) - \mathrm{dt}(P')\big) - \frac{\mathrm{cons}(P') - \mathrm{cons}(P)}{\mathrm{cf}_{\max}} \geq \mathrm{tt}_P \,.$$

This time value cannot be less than tt_P, since tt_P is the minimal travel time when using P. Next we use that $\mathrm{tt}_P - \mathrm{tt}_{P'} > 0$ holds per assumption, which leads to

$$\begin{aligned}
&\mathrm{dt}(P) - \mathrm{dt}(P') - \frac{\mathrm{cons}(P') - \mathrm{cons}(P)}{\mathrm{cf}_{\max}} \geq \mathrm{tt}_P - \mathrm{tt}_{P'} > 0 \\
\Leftrightarrow\;\; &\mathrm{dt}(P) + \frac{\mathrm{cons}(P)}{\mathrm{cf}_{\max}} > \mathrm{dt}(P') + \frac{\mathrm{cons}(P')}{\mathrm{cf}_{\max}} \\
\Leftrightarrow\;\; &\omega(P) > \omega(P'),
\end{aligned}$$

which is a contradiction to P being minimal under ω.

Case 2. Now we consider the case of $\mathrm{dt}(P') > \mathrm{dt}(P)$ (note that $\mathrm{dt}(P') = \mathrm{dt}(P)$ cannot occur since P and P' are non dominating). In this case we also have $\mathrm{cons}(P') < \mathrm{cons}(P)$.

As before, we consider the fastest s-t-path with total travel time $\mathrm{tt}_{P'}$, which contains P'. Again we replace P' with P in this path, which reduces the overall travel time to t by $\mathrm{dt}(P') - \mathrm{dt}(P)$, while it also decreases the b at v by $\mathrm{cons}(P) - \mathrm{cons}(P')$.

This might renders the path infeasible, because the SoC at v might no longer be sufficient to reach t. But the fourth condition ensures that there is a charging station available on the path, which we can use to recharge the missing energy. Furthermore, the third condition states that no over-charging occured between u and this charging station. Therefore, when reaching the charging station, we are still missing a SoC of $\mathrm{cons}(P) - \mathrm{cons}(P')$, compared to the original s-t-path using P'. We now use this charging station to recharge the missing SoC, which ensures that the path to t is again feasible. Altogether the travel time for reaching t when replacing P' with P is given by

$$\mathrm{tt}_{P'} - \big(\mathrm{dt}(P') - \mathrm{dt}(P)\big) + \frac{\mathrm{cons}(P) - \mathrm{cons}(P')}{\mathrm{cf}_{\max}} \geq \mathrm{tt}_P \,.$$

As in the first case, this is equivalent to $\omega(P) > \omega(P')$, which is a contradiction to P being minimal under ω. Thus, our initial assumption must have been false which proves the theorem. □

6.2.1 Constant and Global Charging Speed

We now use Theorem 6.1 to pick one entry from each shortcut's Pareto-set, for the case that the maximal available charging speed is given by $\mathrm{cf}_{\mathrm{max}}$. Recall that the value $\mathrm{cf}_{\mathrm{max}}$ is also used to compute the potential function π_2 (Chapter 5.1.1). Thus this heuristic is well suited for being used together with π_2 as potential function.

Every time a shortcut edge should be relaxed, we search for the entry in the associated Pareto-set, which has the minimal value for ω. Only this entry is then used in the search algorithm, all other Pareto entries are ignored. Since the value of ω does not depend on the current SoC, it is possible to compute the optimal entry for every Pareto-set in a preprocessing step.

As we do not check if all conditions of Theorem 6.1 are satisfied, we cannot guarantee that the resulting algorithm is correct, thus it is only a heuristic. We can, however, try to fulfill some of the conditions. One of the requirements is that the target t is not reachable without recharging. When computing the potential function for the A* search, we also compute a lower bound for the energy required to reach t. This enables us to check if the current SoC might be insufficient to reach t every time an edge from u to v has to be relaxed. We only apply our heuristic if the current SoC at vertex u is less than the minimal SoC required to reach t from u.

We do not check whether the third and fourth condition are satisfied, i.e., if no over-charging occurs and if sufficiently many charging stations are available, all providing a charging speed of $\mathrm{cf}_{\mathrm{max}}$. Our assumption is, that these requirements are often satisfied by chance.

Using this heuristic makes relaxing a shortcut from u to v much more efficient. Now only one entry from the shortcut's Pareto-set has to be relaxed, instead of all of them. This leads to the creation of only one new label for the vertex v, which also reduces the overall number of labels that have to be settled. Furthermore, if only one new label for the vertex v is created, we only have to check for this single label if it is dominated by another label of the vertex v, or if itself dominates some of v's labels.

6.2.2 Lower Bound Potential Functions

The heuristic proposed in the last section requires the knowledge of the maximal available charging speed $\mathrm{cf}_{\mathrm{max}}$. In this section, we introduce a slight modification of our heuristic, where we replace the maximal available charging speed by travel-time-functions. Thus this heuristic is particularly well suited for being used together with π_3 as potential function (Chapter 5.1.3).

We try to avoid using the maximal charging speed $\mathrm{cf}_{\mathrm{max}}$, since it is highly inaccurate for networks which feature different kinds of charging stations like regular charging stations and battery swapping stations. The heuristic presented in the previous section used $\mathrm{cf}_{\mathrm{max}}$ in order to convert energy consumption into travel time. The SoC-function of the vertex currently getting settled as well as the potential function for this vertex also contain information about the available charging speed.

A SoC-function arises from the last seen charging station on the current path. Thus its slope states the charging speed offered by this charging station. When computing the potential functions, travel-time-functions are used. These travel-time-functions are the result of a backwards search form t to the current vertex, thereby they incorporate information about the charging stations which could be used on the remaining path to t.

We use both functions together, in order to find an alternative for ω, that does not use $\mathrm{cf}_{\mathrm{max}}$. Consider the case that a shortcut from u to v should be relaxed. We define a new function ω', which takes the current SoC-function $b(\cdot)$ for the vertex v and one path P which is represented by an entry of the shortcut's Pareto-set, as arguments. We then use the potential function of the vertex v, in order to evaluate the minimal travel time required to reach t if we would use the path P. Thus ω' is defined as

$$\omega'(b, P) := \min_{\mathrm{tt}_v \geq 0} \mathrm{tt}_v + \pi(v, (b \circ P)(\mathrm{tt}_v)),$$

where P is an u-v-path and $b(\cdot)$ is the current SoC-function for the vertex u. The definition minimizes the sum of the travel time tt_v required to reach v, including recharging at the last seen charging station, and the minimal time to reach t from v with a SoC of $(b \circ P)(\mathrm{tt}_v)$. Here $(b \circ P)(\mathrm{tt}_v)$ is the SoC at which the vertex v is reached if it took us a time of tt_v to reach v, including recharging.

Now we evaluate ω' for every path represented by the shortcut's Pareto-set. Afterwards, our search algorithm relaxes only the path which had the minimal value of ω', all other Pareto-entries are ignored. Since the value of ω' depends on the current SoC-function $b(\cdot)$ and the potential function π for the vertex v, it is not possible to determine the optimal Pareto-set entry in a preprocessing step, as it was possible for the heuristic described in the previous section.

In order to compute ω' for a path P, the path has to be relaxed. Thus using this heuristic does not decrease the number of paths that have to be relaxed, as opposed to the heuristic of the last section. But these relaxations are only used to compute the value of ω'. Only one path is relaxed in order to obtain a new label, thus the overall number of labels which are created during the relaxation of a shortcut, as well as the number of dominance checks that have to be performed is equivalent to the heuristic from the previous section.

6.3 Approximating Pareto-Sets

During the CH preprocessing we already used a heuristic witness search, in order to decide if a shortcut is needed or not. The heuristic used for this search can also be applied to our charging function propagating search algorithm. In order to reduce the size of the label-set during the search we introduced a fixed limit for their size. If the size of a label-set exceeds this limit, some of the contained labels are simply deleted.

Instead of deleting arbitrary labels, we decide that the label with the minimal travel time should never be removed, because it potentially leads to the fastest feasible path to t. We also do not delete the label with the highest SoC, because if the target is reachable by any of the label-set's labels, then it is also reachable by the label with the highest SoC. So keeping this label guarantees, that we find a feasible path to t, if there exists one.

Apart from that we would like to delete labels which are most probably not part of a fast solution. We can use the potential π in order to estimate the remaining travel time to the target, similar to the way we used π in the definition of ω'. Given a label for the vertex v represented by the SoC-function $b(\cdot)$, an estimation for the overall travel time to t is given by

$$\min_{\mathrm{tt}_v \geq 0} \mathrm{tt}_v + \pi(v, b(\mathrm{tt}_v)).$$

If we have to delete a label from a label-set we simply delete the one with the maximal estimated travel time for reaching t.

7. Evaluation

In this chapter we evaluate our implementations of the introduced algorithms. First we describe the input data used for the experiments, as well as the general experimental setup. Afterwards, we evaluate the performance of the algorithms introduced in Sections 4, 5, and 6.

7.1 Experimental Setup and Input Data

First we introduce the input used for our experiments. In order to create a complete problem instance we need a road network together with edge weight functions for driving time and energy consumption. Furthermore, we need the locations of charging stations as well as their charging functions.

7.1.1 Road Networks

We evaluate our algorithms using two different types of road networks.

PTV. One of the road networks we use, is kindly provided by the PTV AG for scientific use. Besides information about road segments and intersections, the data includes geographical coordinates and road types. For every road type an average driving speed is given. This information is used together with the length of the edges in order to obtain driving times. We extracted three graphs of different sizes from the provided data, namely *europe*, *germany* and *luxembourg*. The size of these graphs is stated in Table 7.1.

OSM. As a second source for road networks we use OpenStreetMap[1] (OSM) data. Sabine Storandt kindly provided us the graph of southern Germany (called *osm-sger*), which was used in the eperimental section of [Sto12b]. The graph covers the part of the OSM graph of Germany which is located south of a latitude of 48.9. In addition to that we extracted the complete graph of Germany (called *osm-ger*) from OpenStreetMap. The data of these road networks again contains geographical coordinates and to some extend information about the road type. As before this is used to obtain driving times for the edges.

7.1.2 Energy Consumption

We already have the graphs together with driving times, but we also need an edge weight function for the energy consumption. An EV's energy consumption depends strongly on

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

Graph	#vertices (n)	#edges (m)	#edges with cons ≤ 0	
germany	4 692 091	10 805 429	1 119 710	(10.36%)
europe	26 316 863	60 614 720	7 116 296	(11.74%)
luxembourg	36 457	81 950	12 947	(15.79%)
osm-sger	5 588 146	11 711 088	1 142 391	(9.75%)
osm-ger	20 690 320	41 791 542	1 735 554	(4.15%)

Table 7.1: The different graphs used in the evaluation section.

the road slope. In order to compute the slope we use elevation data from the Shuttle Radar Topography Mission (SRTM), which is freely available from the CGIAR Consortium for Spatial Information[2]. The data covers large parts of the world with a precision of three arc seconds, which corresponds to 90 meters at the equator. Parts of the graphs which are not covered by this data are removed. In order to compute the actual energy consumption we use two models depending on the type of the graph.

For the graph originating from PTV data, the energy consumption is computed using PHEM (Passenger car and Heavy duty Emission Model) [HRZL09], which is developed by the Graz University of Technology. Among others, the model can be used to compute an EV's energy consumption for a great variety of driving situations, taking road categories, driving speed and slope into account. The road types provided by the PTV data were mapped to PHEM road categories using a heuristic previously used by Baum et al. [BDPW13]. PHEM models energy consumption for many EV configurations and vehicle types. For our experiments we choose the model of an Peugeot iOn. This vehicle is equipped with a 16 kWh battery, which will also be used in our experiments. Additionally we use a fictive vehicle which has the same energy consumption but features a larger 60 kWh battery.

For the OSM based graphs we use the consumption model introduced in [EFS11]. The same model was used in [Sto12b], thus together with the osm sger graph we obtain the same setting that was used in their experimental section. In their model, the energy consumption for an edge $e = (u, v)$ is given by

$$\begin{aligned}\text{cons}(e) &:= \text{fixed}(e) + \begin{cases}\eta(v) - \eta(u) & \text{if } \eta(v) - \eta(u) \geq 0\\ \alpha \cdot (\eta(v) - \eta(u)) & \text{else}\end{cases}\\ \text{fixed}(e) &:= \text{dist}(u, v)/50\end{aligned}$$

where $dist(u, v)$ is the linear distance between the two vertices in meters, $\eta(u)$ is the height evaluation of a vertex in meters, and α models the EV's capability of recuperating energy when driving downhill, which is set to 0.25 in their and our experiments.

Table 7.1 shows an overview over all graphs used in the experimental section. Besides the number of vertices and the number of edges, the table also shows the number of edges with negative energy consumption according to the used model.

7.1.3 Charging Stations

In addition to the road networks with driving time and energy consumption we also need the locations of charging stations. We will evaluate our algorithm on different sets of charging stations. For a realistic setting we use charging station locations from ChargeMap[3], which lists a huge amount of charging stations all over the world. Their data contains geographic

[2]http://srtm.csi.cgiar.org/
[3]http://chargemap.com/

coordinates. We use these coordinates to map the charging stations to the vertex in our graph which is closest. If the distance to the closest vertex exceeds 20 meters we ignore the charging station.

Additionally, we evaluate our algorithms on sets of randomly chosen charging stations similar to the experimental setting of [Sto12b]. The authors of [SF12] observed that the number of charging stations in a road network is typically $\mathcal{O}(\sqrt{n})$, where n is the number of vertices in the graph. Therefore, we use this as an orientation for the size of the random charging station sets.

7.1.4 Charging Functions

The last input data we need are the charging functions, which depend on many factors, like station type, battery capacity, current SoC, and desired SoC. One type of charging stations are battery swapping stations. At such a station it is possible to swap the entire battery in about 90 seconds. We assume that in addition to the swapping time one would also lose time due to reaching the actual station and paying afterwards. We take this into account and model the according charging function in such a way that the complete recharging process always takes three minutes.

The charging function for regular charging stations are a bit more complex. Their shape depends on the battery capacity and the available power. Using a normal electrical outlet at home yields a power of 3.7 kW. Most charging stations in use today met the IEC 62196 typ 2 standard and yield a power of 11 kW or 22 kW. Supercharger stations by Tesla yields a power of up to 120 kW, whereby the maximal reachable SoC is limited to 80%.

In order to model charging functions for this kind of stations we use interpolation points kindly provided by Martin Uhrig et al., which have developed a model for charging function based on real measurements of Lithium-Ion-Batteries [UL13]. The data contains a table for every combination of battery capacity and available power, which states the charging time for given initial and desired SoC (An example table for the battery capacity $M = 16\,\mathrm{kWh}$ and an available power of 11 kW is shown in the appendix).

The data shows nicely that the charging process is linear up to 80%. Furthermore, it becomes evident that our assumption $\mathrm{cf}(b, \mathrm{ct}) := \mathrm{cf}'(\mathrm{ct} + \mathrm{cf}'^{-1}(b))$ from Chapter 3.1 is reasonable.

7.2 Experiments

In this section we evaluate the performance of our algorithms in detail. We implemented all our algorithms in C++ using `g++` version 4.7.1 (64 bit) with optimization level 3 as compiler. The experiments were performed on a single core of a machine with a dual 8-core Intel Xeon E5-2670 processor clocked at 2.6 GHz, with 64 GiB of DDR3-1600 RAM. In order to ensure reproducibility of the experiments only one algorithm was running at any given time.

7.2.1 CH Preprocessing

We start by evaluating the CH preprocessing. We compute the CH as described in Chapter 5.2 with a heuristic vertex ordering. In order to determine the next vertex for contraction, we use a linear combination of all three introduced criteria, which weights the *edge difference* criterion 16 times heavier than the *contracted neighbors* criterion and the *search space depth* criterion. Furthermore we use an upper bound of 20 for the search depth of the used witness search, and an upper bound of 10 for its label-set size.

Table 7.2: Computation times of the CH preprocessing on various graphs and for different core sizes. All running times are in the format h:mm:ss. Missing values indicates that the preprocessing had not terminated after eight hours. Except for the graph of Luxembourg, where some values are missing because the number of charging stations, which is the minimal core size, is greater than the desired core size. CS denotes the used charging station set. The last column states computation times for a core that only contains the charging stations.

			Core size			
Graph	CS	Capacity	1%	0.5%	0.2%	only CS
germany	cm-1966	16 kWh	6:56	13:26	2:30:46	?
		60 kWh	7:35	14:58	4:49:02	?
	r-2166	16 kWh	7:32	14:13	3:35:14	?
		60 kWh	8:17	15:22	7:21:37	?
	r-4332	16 kWh	7:51	17:03	?	?
		60 kWh	8:41	18:15	?	?
	∅	16 kWh	6:53	12:10	56:30	?
europe	cm-15666	16 kWh	1:11:05	3:38:29	?	?
		60 kWh	1:15:49	3:48:11	?	?
	r-5579	16 kWh	1:02:13	3:12:42	?	?
		60 kWh	1:04:47	3:26:34	?	?
	r-11158	16 kWh	1:37:59	5:27:31	?	?
		60 kWh	1:41:18	5:59:57	?	?
lux	cm-36	16 kWh	4	5	11	37
		4 kWh	1	1	3	5
	r-190	16 kWh	5	—	—	9:47
		4 kWh	5	—	—	38
	r-280	16 kWh	22	—	—	26:10
		4 kWh	14	—	—	1:20
osm-sger	cm-643	~100 km	8:34	10:43	15:43	7:54:42
		~500 km	8:48	11:01	17:04	?
	r-100	~100 km	8:14	10:13	14:19	1:09:29
		~500 km	8:56	10:58	15:04	1:42:37
	r-2363	~100 km	8:41	11:07	21:24	?
		~500 km	9:14	11:45	25:24	?
	∅	~100 km	8:06	10:07	14:15	53:20
osm-ger	cm-2126	~100 km	8:03	9:06	13:05	7:37:50
		~500 km	8:49	9:55	14:18	?
	r-4548	~100 km	8:56	10:17	15:38	?
		~500 km	9:12	10:48	17:13	?

We tested the preprocessing on all graphs with varying battery capacities and charging station sets. The results are shown in Tables 7.2 and 7.3. The column labeled CS describes the used set of charging stations, where cm stands for data from ChargeMap and r for a random set. The consecutive number specifies the number of charging stations used. For comparison only, we have also performed the preprocessing using an empty set of charging stations. While this does not yield a useful graph for our algorithm, it can be used to analyze the impact of the charging stations. Recall that charging stations influence the preprocessing, since it is forbidden to contract them.

For the PTV graphs we used battery capacities of 16 kWh and 60 kWh, except for Luxembourg graph. Because of this graph's size, a 16 kWh is nearly sufficient to reach every vertex without recharging. Thus we used a small 4 kWh battery, so that using a charging station becomes necessary on more routes. For the OSM graphs we used battery capacities which are equivalent to the energy consumption of driving 100 km or 500 km on flat ground.

Table 7.2 states the times needed to compute CHs up to a certain core size for the different input graphs and charging station sets. We stopped the computation after eight hours if it was not already finished, thus some values are missing in the table. In the case of the Luxembourg graph, some core sizes are not possible because there are more charging stations then the size of the desired core and it is not allowed to contract charging stations.

The data shows, that computing a CH until the core contains only 1% of the vertices, is for all graphs possible and does not take to much time. Furthermore we see, that as the desired core size decreases, the computation time quickly increases. Contracting further vertices becomes finally so slow that is unpractical for most graphs to compute a CH until only charging stations remain uncontracted.

The computation times show that a larger charging station set increases the time needed to reach a certain core size. For a core size of 1% the computation times differ only by a few minutes. But as the core gets smaller, the impact of a larger charging station sets quickly becomes visible. For an example we can look at the germany graph with a 16 kWh battery. Here computing a core containing 1% of the vertices takes between six and eight minutes independent of the charging station set. Computing a core containing 0.2% for the charging station set from ChargeMap takes 2.5 hours. Computing a core of the same size for a slightly bigger set of random charging station takes already one additional hour. If we double the number of random charging stations, it is no longer possible to compute the core in under eight hours.

We also observe, that although the osm-sger and osm-ger graph have more vertices than the germany graph, it is possible to compute smaller core graphs for osm-ger and osm-sger. On reason for this could be that these graphs have an initially smaller average degree. While the germany graph has an initial degree of 4.61, the OSM graphs have only average degrees of 4.19 for osm-sger and 4.03 for osm-ger. Furthermore the different consumption model used for the OSM graphs could be reason for the faster preprocessing time.

Next we analyze why the CH preprocessing becomes so slow for core size below 1% of the original vertices. While the fact that we do not allow to contract charging stations, has certainly an impact on the computation time, the presence of charging stations can be ruled out as reason for the slow computation times. For the germany and osm-sger graph we have additionally included data showing the computation time for the CH preprocessing if no charging station exist. The preprocessing is indeed faster if no charging stations exist, but the computation time still increases drastically for smaller core sizes. For the germany graph we are unable to compute a complete CH even if no charging stations are given.

In order to understand the times needed for the preprocessing we can look at Table 7.3. This table states the average vertex degree for all combination of input graphs and core sizes. Since the shortcuts added during preprocessing may represent several Pareto optimal paths, we slightly modify the definition of vertex degree. Each of the entries in the Pareto-set associated with a shortcut could also be represented using a multi-edge. In order to account for this we weight each shortcut edge with the size of its Pareto-set when computing the vertex degree.

As shown in Table 7.3 the average vertex degree increases quickly with decreasing core size. A higher vertex degree increases directly the preprocessing time. Every time a vertex get contracted we have to check for every pair of incoming and outgoing edge, if a shortcut is

Table 7.3: Result of the CH preprocessing on various graphs and for different core sizes. The table states the average vertex degree in the core graph. When computing the vertex degree, we weight shortcuts by the size of their associated Pareto-sets.

			Core size			
Graph	CS	Capacity	1%	0.5%	0.2%	only CS
germany	cm-1966	16 kWh	52.53	233.57	5 266.13	?
		60 kWh	54.30	235.14	5 456.68	?
	r-2166	16 kWh	54.59	240.55	6 737.36	?
		60 kWh	55.13	252.37	15 849.15	?
	r-4332	16 kWh	60.38	301.30	?	?
		60 kWh	62:94	324.56	?	?
	∅	16 kWh	51.57	194.40	2 387.39	?
europe	cm-15666	16 kWh	64.75	280.96	?	?
		60 kWh	64.83	281.27	?	?
	r-5579	16 kWh	61.60	273.55	?	?
		60 kWh	61.68	274.08	?	?
	r-11158	16 kWh	75.95	358.93	?	?
		60 kWh	77.21	361.38	?	?
lux	cm-36	16 kWh	110.53	129.03	327.52	503.06
		4 kWh	78.35	98.78	123.62	273.84
	r-190	16 kWh	163.92	—	—	1 694.03
		4 kWh	147.09	—	—	491.28
	r-280	16 kWh	473.84	—	—	3 404.60
		4 kWh	362.77	—	—	1 047.01
osm-sger	cm-643	~100 km	13.71	23.10	146.89	17 803.30
		~500 km	14.24	23.25	151.23	?
	r-100	~100 km	13.95	23.17	127.30	6 448.64
		~500 km	14.18	23.21	128.38	7 207.75
	r-2363	~100 km	14.45	26.17	433.97	?
		~500 km	14.58	26.20	436.76	?
	∅	~100 km	14.07	23.38	126.70	9 084.26
osm-ger	cm-2126	~100 km	10.78	18.54	119.52	9 555.46
		~500 km	10.84	18.84	119.93	?
	r-4548	~100 km	11.01	19.36	122.81	?
		~500 km	11.16	19.84	125.30	?

needed. In the case that these edges are already shortcuts, we have to check for every pair of one Pareto entry from the incoming shortcut and one Pareto entry from the outgoing shortcut, if a new shortcut is needed. Thus the number of witness searches increases quadratic with the average vertex degree.

Although changing the set of charging stations influences the average vertex degree, it is not the reason for the increasing vertex degree as the core gets smaller. This becomes particularly evident when we look at the average vertex degrees for the CH preprocessing of the germany and osm-sger graphs without any charging stations. Here the vertex degree still increases as the core gets smaller. The reason for this is of course, that each shortcut has to represent all Pareto optimal paths between its head and tail vertex. But the number of Pareto optimal paths can be exponential in the size of the original graph.

Table 7.4: Impact of the Core size on the query time of our algorithms in seconds. For this experiment we used the germany graph with regular charging stations offering a charging power 22 kW. The table states average running times in seconds. For each combination of core size and algorithm we evaluated the same 200 random queries.

		Core size				
A*	Heuristic	2%	1%	0.5%	0.25%	0.2%
π_2	×	26.41	32.20	57.72	118.13	164.47
π_2	✓	0.39	0.17	0.12	0.35	0.96
π_3	×	6.83	5.24	8.95	24.07	39.51
π_3	✓	3.86	1.79	3.98	6.34	8.97

Furthermore the data presented in Table 7.3 shows once again a difference between the PTV and the OSM graphs. The average vertex degree for the OSM graphs is much lower than the degree for the PTV graphs. Because of this it is possible to compute smaller cores for the OSM graphs. The consumption model used for the OSM graphs could be a reason for the smaller average degrees.

7.2.2 Impact of the Core Size on the Algorithms

As we observed in the previous section, it is not possible to contract all vertices during the CH preprocessing. Thus at some point the CH preprocessing has to be aborted. We now evaluate at which point further vertex contractions are no longer meaningful. We say that further vertex contractions are not meaningful, if the time need for the contraction cannot be justified with the speedup gained for the query algorithm.

We evaluate the running times for some of our query algorithms depending on the size of the core graph. For this experiment we use the germany graph, together with the charging stations from ChargeMap and a battery capacity of 16 kWh. Table 7.4 shows the resulting query times in seconds. For each combination of core size and algorithm we performed the same 200 random queries.

The resulting data shows for all algorithms an increasing query time below a core size of 0.5%. When using π_3 as potential function minimal query times are achieved when using the CH with a core size of 1%. If π_2 is used as potential function, then the query time is minimized for a core size of 0.5% or 2% depending on whether the heuristic is used or not.

In order to understand why the query times increase with decreasing core this, after the core size drops below a certain value, we analyze the behavior of the query algorithm in greater detail. For this purpose we focus on the exact algorithm using π_3 as potential function.

Table 7.5 shows, besides average query time, also the average size of the label-sets created during the search, as well as the average number of settle and relax operations. The data shows that the number of settle operations decreases with decreasing core size. This was expected, since reducing the size of the core also reduces the size of the query algorithms search space.

But the decreased number of settle operations does not lead to a decreased query time. The reason for this is the same that already prevented us from computing a complete CH, the average core degree. As shown in the table, the average core degree increases from about 19 for a core containing 2% of the original vertices up to over 5 200 for a core containing 0.2%

Table 7.5: This table shows the performance of the exact algorithm using π_3 as potential in more detail. The experimental setup is the same as the one for Table 7.4

Core size	2%	1%	0.5%	0.25%	0.2%
Query time [s]	6.83	5.24	8.95	24.07	39.51
Core degree	18.85	52.53	233.57	1 987.37	5 266.13
Label-set size	121.77	57.54	25.55	11.50	10.86
Settle operations	137 688	72 086	36 094	18 655	16 206
Relax operations	1 230 682	1 864 685	4 285 947	14 760 137	27 871 790

of the original vertices. But a higher degree means that more edges, or Pareto entries of a shortcut, have to be relaxed, which leads to the high number of relax operations as they can be seen in the table. At some point the time lost due to additional edge relaxations is greater then the time saved be the smaller search space. At this point is clearly not meaningful to contract further vertices, since it increases both, the preprocessing time and the query time.

Since query times are minimal for a core size of round about 1%, we will use A Ch with this core size for all further experiments.

7.2.3 Speedup Techniques

In this section we evaluate the speedup techniques we introduced in Chapter 5. First we will compare the different Potential functions we proposed. Afterwards evaluate the speedup that can be achieved using A* or CHs ore both techniques combined.

Potential Functions. The potential functions we proposed can be divided in two. Th first group are potential function which are computed using labels of constant size for the backwards search. The potential functions π_1 and π_2 belong to this group. With these functions we could only use global information about the charging stations, like the maximal charging speed. Since this might not be accurate if several different types of charging stations are available, we introduced a second class of potential functions which are computed using lower bound functions as labels for the backwards search.

Now we want to analyze how these two types of potential functions perform, given different sets of available charging stations. For our experiment we use two types of charging stations that differ significantly, namely battery swapping stations and regular charging stations with a charging power of 11 kW. Using the battery swapping station it takes three minutes to charge the battery up to 100%. Using the regular charging station it takes a bit more than two hours to charge a 16 kWh battery up to 100%.

We start with a set of charging stations that only contains regular charging stations. Then we gradually increase the proportion of battery swapping stations and observe how the average query times change. As soon as only one battery swapping station is contained in the charging station set, we have to use the charging speed of this battery swapping station during the computation of π_2.

The results of this experiment can be seen in Figure 7.1. Each data point is the average query time of the same 200 random queries, which where used for every data point. We used the germany graph with charging station locations from ChargeMap for this experiment. The type of each charging station was chosen randomly according to the required ratio of battery swapping stations and regular stations.

Unfortunately the exact algorithm using π_2 as potential was two slow to perform the full experiment. Because of this we evaluated only 100 queries for this algorithm and interrupted the computation after 20 minutes if no solution was found.

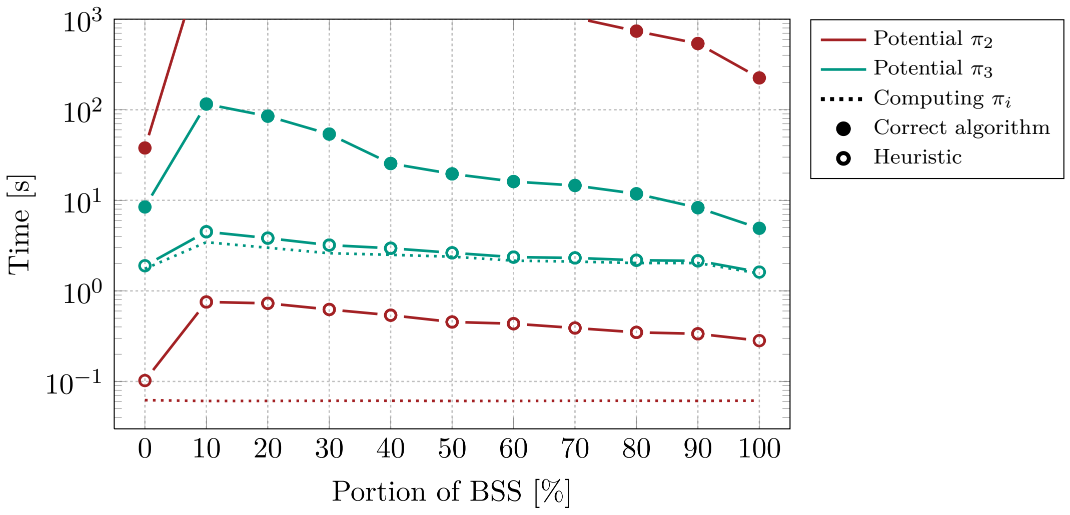

Figure 7.1: Running times of our algorithm for the germany graph. The x-axis denotes the portion of battery swapping stations, the other charging stations are regular station offering a power of 11 kW.

The figure shows the average query times for the potentials π_2 and π_3. Furthermore we evaluated both, the exact as well as the heuristic algorithm. Additionally the figure contains also the time needed to compute the values for the potentials only, which is represented by the dotted lines. Clearly visible there is a sudden increase in the query time as the first battery swapping stations are made available. This matches exactly our prediction. Furthermore we see that the exact algorithm using π_3 as potential is always faster than the one using π_2.

For the heuristic version of out our algorithm we can see that that using π_2 yields a better query time, independent of the composition of the charging stations. The main reason for that is that the query time of the heuristic algorithm using π_3 is dominated by the time needed to compute the potential.

General Comparison of all Speedup Techniques. Now we want to evaluate the general speedup that can be achieved using our algorithms. Since our base algorithm without any speedup techniques is rather slow, we perform this experiment on the lux graph. Because we do not want that all vertices are reachable without recharging, we use a small 4 kW battery. As we have seen, a low number of battery swapping stations together with many slow charging stations is the worst case scenario for our algorithms, therefore we us a charging station set containing 10% swapping stations and 90% regular stations for this experiment. As charging station locations we again use the locations from ChargeMap.

Table 7.6 shows the average query times for all combinations of speedup techniques as well as the base algorithm. Additionally we report the number of settle and relax operations, as they provide further insight into the performance of the algorithms.

The query time of the base algorithm can be seen in the first row of the table. It takes slightly more than one second to compute the optimal feasible path for the given experimental setting. If we only use A* as speedup technique we achieve a speedup between 1.5 and 8.9 depending on the used potential function. Using a CH without additional techniques we achieve a speedup of 23.6.

When using a CH we observe that especially the number of settled operation is reduced. Using the base algorithm, an average of over 400 000 settle operations had to be performed. This number is reduced to about 1 600 when using a CH. The reason for this is the reduced

Table 7.6: Impact of the speedup techniques CH and A* on our algorithm. Every value is the average query time of the same 1000 random queries. The battery capacity is 4 kW, 10% of the available charging stations were swapping stations, the other stations were regular once, offering a power of 11 kW.

CH	A*	time [ms]	speedup	settle operations	relax operations
×	—	1 025.9	1	437 174	590 545
×	π_1	135.1	7.5	58 304	80 370
×	π_2	114.7	8.9	47 732	65 623
×	π_3	676.1	1.5	98 383	132 569
✓	—	43.2	23.6	1 640	124 944
✓	π_1	21.1	48.5	750	36 824
✓	π_2	17.3	58.9	642	24 658
✓	π_3	15.8	64.9	701	22 352

Table 7.7: Speedup and quality achieved by our heuristic algorithms, which we introduced in Chapter 6.2. The experimental setup is the same as for Table 7.6. The algorithms based on the charging function propagating algorithm always found a feasible path, if on existed. The conventional car drivers approach only managed to find a solution in 86% of the queries where a feasible solution exists.

CH	A*	time [ms]	speedup	settle operations	relax operations	quality
×	—	518.4	1.9	282 579	381 814	1.00
×	π_1	84.0	12.2	42 910	58 672	0.99
×	π_2	73.4	13.9	34 997	47 734	0.99
×	π_3	683.1	1.5	98 383	132 569	0.99
✓	—	4.0	251.9	199	2 307	0.98
✓	π_1	8.5	120.1	109	1 690	0.97
✓	π_2	8.3	123.0	101	1 035	0.97
✓	π_3	13.3	77.1	658	22 009	0.98
CCDA		4.5	225.5	170	538	(0.92)

size of the search space when using a CH. Compared to the number of settle operations, the number of relax operations is only reduced by a factor of 5. The reason for this is, that the CH preprocessing increases the average vertex degree in the core graph.

The maximal speedup can be achieved when combining CH and A* search. In this case using π_3 as potential reduces the average query time to 15.8 ms, which is nearly 65 times faster than the base algorithm.

Speedup of Heuristic Algorithms. We can analyze the speedup of our heuristic approaches in the same way as we did for the exact algorithms. We use the same experimental setup as well as the same random queries, which we used to evaluate the speedup of the exact algorithms, in order to evaluate the our heuristics. The results are shown in Table 7.7.

Besides the achieved speedup we are now also interested in the quality of the computed solutions, since the heuristics do not guarantee to find the optimal solution. We use the quotient of the minimal travel time and the travel time of the solution computed by the heuristic as measurement for the quality of the solution. This means that a quality of one indicates that the optimal solution has been found. A solution which is slower then the optimum yields a quality less than one.

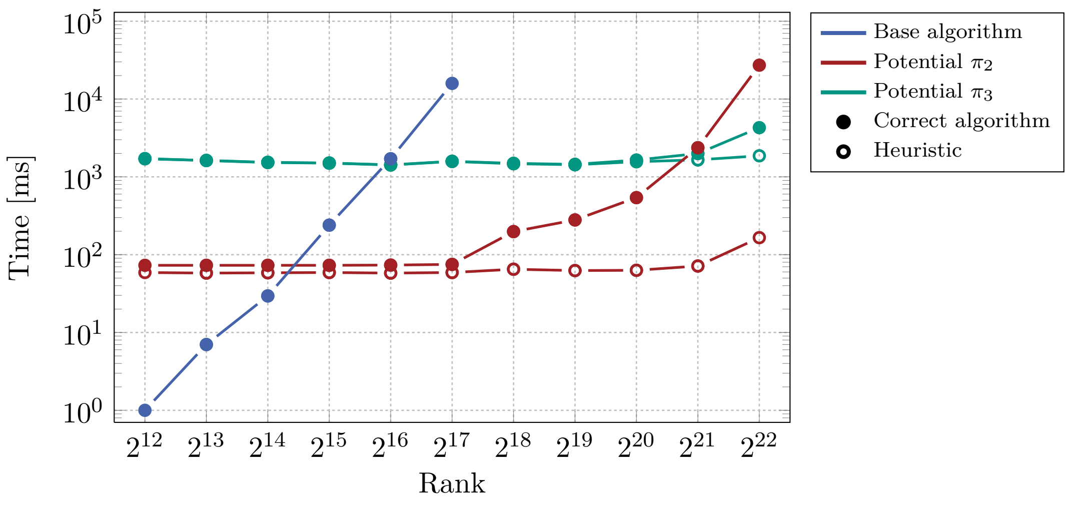

Figure 7.2: Running times of our algorithm, using a 16 kWh battery, depending on the Dijkstra rank of the query. We evaluated a total of 10 000 random queries per algorithm on the germany graph for this experiment. Since the base algorithm is rather slow, we interrupted its computation if no solution was found after two minutes, thus we report no query times for a Dijkstra rank greater than 2^{17}.

Furthermore it is possible that a heuristic does not find any solutions, although there exists a feasible path. In our experiment, the heuristics based an the charging function propagating algorithm always found a feasible solution, if one existed. The conventional car drivers approach on the other hand could only find a solution in 86% of the cases where a feasible path existed. The quality reported for this algorithm in Table 7.7 only considers the cases, in which a feasible path was found by the heuristic.

Besides the rather poor quality of the found solutions, the conventional car drivers approach yields a relative high speedup of 225.5 compared to the exact base algorithm. This speedup corresponds to an average query time of 4.5 ms. Using only a CH as speedup technique yields the minimal average query time of only 4 ms. Using a combination of CH and A* can reduce the number of settle and relax operations compared to using only a CH. This does not lead to a further reduction of the query time, because computing the potential function takes to much additional time. in Figure 7.1 we have already seen that computing the potential can actually dominate the overall query time when using the heuristic.

Speedup depending on Query Distance. Next we want to analyze the query time of our algorithms depending on the distance between the source and the target vertex. As measurement for the distance of a query we use the so called Dijkstra *rank*. The order in which the vertices get settled during an execution of Dijkstra's algorithm can be used to rank the vertices. Recall that Dijkstra's algorithm settles each algorithm at most once, therefore this order is well defined. Since plain Dijkstra is used to define the Dijkstra rank, it is only defined for a single search criterion. For our scenario we use the driving time metric to define the Dijkstra rank, since it is the metric we want to minimize, and energy consumption is only used as restriction. Given a fixed source vertex s we compute the Dijkstra rank of all other vertices using a Dijkstra search starting at s. The Dijkstra rank of a vertex v is than defined as the number of vertices that have been settled before v got settled.

For our experiment we use again the germany graph and charging station locations from ChargeMap. All charging stations in the graph were regular charging stations offering a power of 11 kW. Figure 7.2 shows the median of the query times, if a 16 kWh battery is used.

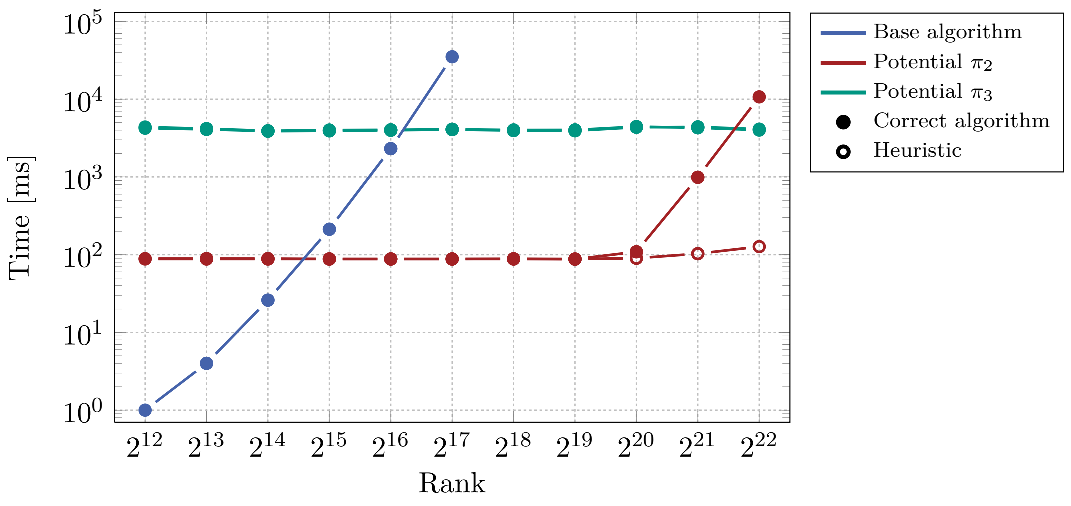

Figure 7.3: Running times of our algorithm, using a 60 kWh battery, depending on the Dijkstra rank of the query. We evaluated a total of 10 000 random queries per algorithm on the germany graph for this experiment. Since the base algorithm is rather slow, we interrupted its computation if no solution was found after two minutes, thus we report no query times for a Dijkstra rank greater than 2^{17}. The query of the heuristic algorithm using π_3 as potential are not visible in this plot because they are nearly the same as for the correct algorithm.

Figure 7.3 shows the median of the query times, if a 60 kWh is Used. In both cases we observe that the query time of the base algorithm quickly increases as the Dijkstra rank rises. For a Dijkstra rank above 2^{17} the base algorithm becomes unpractical and we interrupted the computation if no solution was found after two minutes. Therefore we do not report the query times for a Dijkstra rank above 2^{17}.

Both Dijkstra rank plots show that the query time of our algorithm using A* together with a potential π_2 or π_3 is constant for low ranks. The reason for this is, that the computation of the potential function is independent from the the distance between s and t. For queries with a low rank the computation of the potential function dominates the overall query time, thus the query time is also independent from the query distance for small ranks.

When using a 16 kWh battery, the query time of the exact algorithm using π_2 starts to increase at a rank of about 2^{18}. When using a 60 kWh battery, the query times stays constant until a rank of 2^{20} is reached. This difference can be explained with the extended driving range due to the larger battery. If the potential function indicates that the target is reachable without recharging, the search algorithm will most probably first settle such labels that do not use a charging station. This means that most of the SoC-functions used during the search are constant, which results in low query times. Indeed a Dijkstra rank of 2^{17} roughly corresponds to the driving range of a 16 kWh battery, while a rank of 2^{19} corresponds to the driving range of a 60 kWh battery.

The Dijkstra rank plots show also that the query time of the heuristic algorithms is nearly completely independent from query distance. The reason for this is again, that the query time is dominated by the time needed to compute the potential function.

In the appendix we have included a more detailed version of the Dijkstra rank figures using box plots. These plots show the existence of some outlier queries which take much longer than the median query time. Furthermore the box plots reveal that using π_2 as potential leads to a much higher variation in the query times than using π_3.

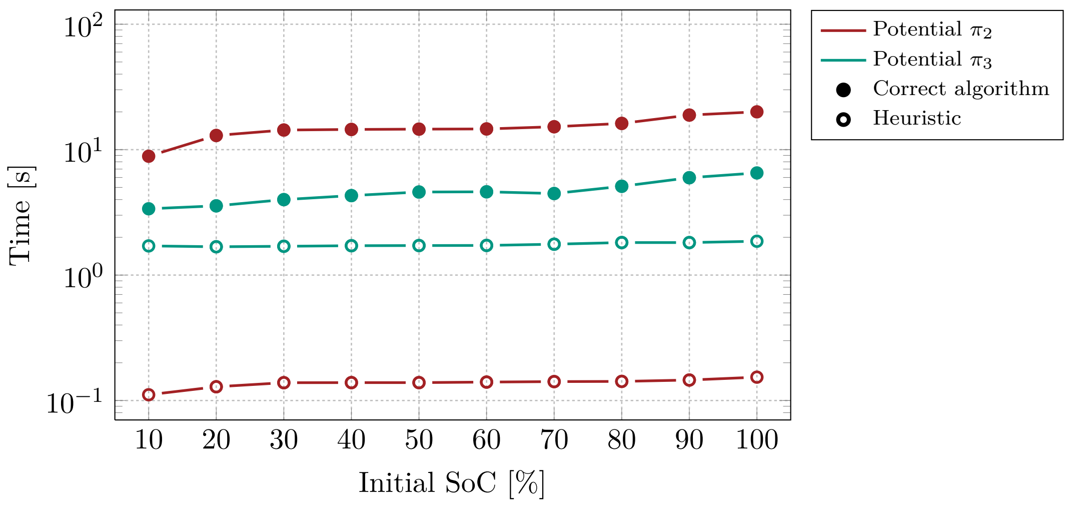

Figure 7.4: Running times of our algorithm for the germany graph with changing initial SoC. For this experiment we used charging station location from ChargeMap and regular charging stations offering 11 kW. For each data point we evaluated the same 200 random queries and computed the average of the query times.

7.2.4 General Evaluation

We conclude our Experimental section with a general evaluation of the query times of our algorithms. First we analyses how the initial SoC of the battery influences the query times. Finally we report average query times for all graphs and different types of charging stations.

Influence of the initial SoC. Figure 7.4 shows how the average query times of our algorithms change depending on the initial SoC. As before we use the germany graph together with charging stations from ChargeMap. Furthermore we used a 16 kWh battery and regular charging stations offering 11 kW. We evaluated the same 200 random queries for every combination of initial SoC and algorithm.

The query time of the individual algorithms changes only slightly, by a factor of two to four, as the initial SoC changes. But the data shows also that query time of all algorithm decreases as the initial SoC increases. The reason for this is, that a high initial SoC enlarges the algorithms search space, because it allows to explore energy wasting paths. The initially given energy might be wasted: As long as no recharging is required to reach the target, it does not matter how much of the initial energy is consumed, because spending more energy does not increase the travel time. If, however, no initial energy is given, then all the consumed energy has to charged first. In this case wasting energy directly increases the driving time. Therefore, if this initial SoC is low, it is more likely that energy wasting paths get dominated by other, more efficient paths early on. This reduces the number of labels that get settled and therefore the query time.

Query times for all graphs. Finally, Table 7.8 states the query times achieved by our algorithm for all graphs we tested and for three different sets of charging stations. For the four graphs osm-sger, osm.ger, lux, and germany we evaluated 1 000 random queries for each algorithm. In addition to the exact algorithms we also include the query times for the heuristic algorithm using π_2 as potential. We do not include the heuristic using π_3 as potential, because our previous experiments show that using π_2 yields always faster query times. For the heuristic we also include the quality of the found solutions, which is given by the quotient of the optimal travel time and the travel time of the heuristic solution. Furthermore we report the percentage of queries for which the heuristic found a feasible path, if one existed.

Table 7.8: Query times of our algorithms for all graphs and different charging station types. For this experiment we used a 16 kWh battery (~100 km driving range for the OSM graphs) and charging station locations from ChargeMap. The reported query times are in milliseconds. For the heuristic algorithm using potential π_2 we also include the quality of the found solutions as well as the percentage of found solutions.

Only battery swapping stations:

Algorithm	π_2	π_3	π_2 with Heuristic		
osm-sger	853.6	1 003.0	111.5	(0.99	100%)
osm-ger	1 381.5	1 496.1	168.2	(0.99	100%)
lux	3.1	8.2	1.7	(1.00	100%)
germany	33 487.0	3 091.5	372.0	(0.99	100%)
europe	—	—	104 479.2	—	

Only regular charging stations offering 11 kW:

Algorithm	π_2	π_3	π_2 with Heuristic		
osm-sger	733.4	1 186.0	141.4	(0.98	100%)
osm-ger	1 063.6	1 880.1	182.1	(0.99	100%)
lux	2.4	9.7	1.1	(0.99	100%)
germany	15 974.3	5 960.5	127.3	(0.99	99%)
europe	—	—	87 943.5	—	

10% BSS and 90% regular stations (11 kW):

Algorithm	π_2	π_3	π_2 with Heuristic		
osm-sger	809.8	679.2	153.0	(0.99	100%)
osm-ger	3 794.5	2 309.6	533.1	(0.97	100%)
lux	3.1	11.3	1.6	(1.00	100%)
germany	57 394.6	38 931.3	1 255.8	(1.00	100%)
europe	—	—	459 323.1	—	

We evaluated our algorithms for three different combinations of charging stations. First, we only use Battery swapping stations (BSS). Restricted to this station type our problem is basically the same as the one considered in [Sto12b]. Our best algorithm for this setting (π_2) has an average query time of 854 ms on the osm-sger graph. This is comparable to the results achieved by the authors of [Sto12b].

In contrast to the algorithm proposed in [Sto12b] our algorithm is of course capable of using various types of charging stations. Thus, after we have tested our algorithms for battery swapping stations, which are the fastest charging stations, we now test our algorithms for slow regular charging stations, which provide a charging power of 11 kWh. Using the slow charging stations improves the query times of the algorithm using π_2 as potential, while it slightly increases the query times of the algorithm using π_3. Once again we observe that π_3 can only outperform π_2 on large graphs, such as the germany graph.

We also report the query times for the case that only 10% of the charging stations are battery swapping stations and the remaining charging stations are slow regular stations. We include this case as our previous experiments show that this is the most difficult setting for our algorithms. Indeed we observe an increased query time for this setting. But our algorithms remain applicable even for medium-sized graphs like the germany graph.

The table states also query times for the heuristic algorithm on the europe graph. Our exact algorithms are not applicable for this graph. Thus we do not report exact query times. Furthermore, as we do not know the minimal travel times, we cannot evaluate the quality of the solutions found by the algorithm. Since the heuristic is rather slow, we only performed 100 random queries for the europe graph.

7.3 Example Queries

In the section we present some exemplary queries and the resulting routes computed by our algorithms. The examples provided in this section should not be used to draw conclusions about the performance of the algorithm. Instead, the examples should be used to get an impression on how the algorithms work and how the different potentials as well as the heuristics influence the algorithm, and its search space.

Figure 7.5 shows the search space of a query from Karlsruhe to Paderborn. The figure compares three algorithms: the exact algorithms using π_2 and π_3 as potential, and the heuristic using π_2 as potential. All vertices which were settled during the search are marked. The brightness of the mark indicates how many labels have been settled for the vertex, the brighter the mark, the more labels have been settled.

Figure 7.6 shows the search space, of the heuristic algorithm using π_2 as potential, for a query from Karlsruhe to Berlin. This example is particularly interesting because the search space splits naturally into branches. This behavior can be observed quite frequently. The reason for this is that the routs lead from one charging station to the next. Thus the location of the charging station limits the search space to some corridors.

Figure 7.7 shows an example for a continental query. We used the heuristic algorithm with π_3 as potential in order to compute a route from Lisbon to Stockholm for an electric vehicle with a 60 kWh battery. It took us slightly more than five minutes to compute this route.

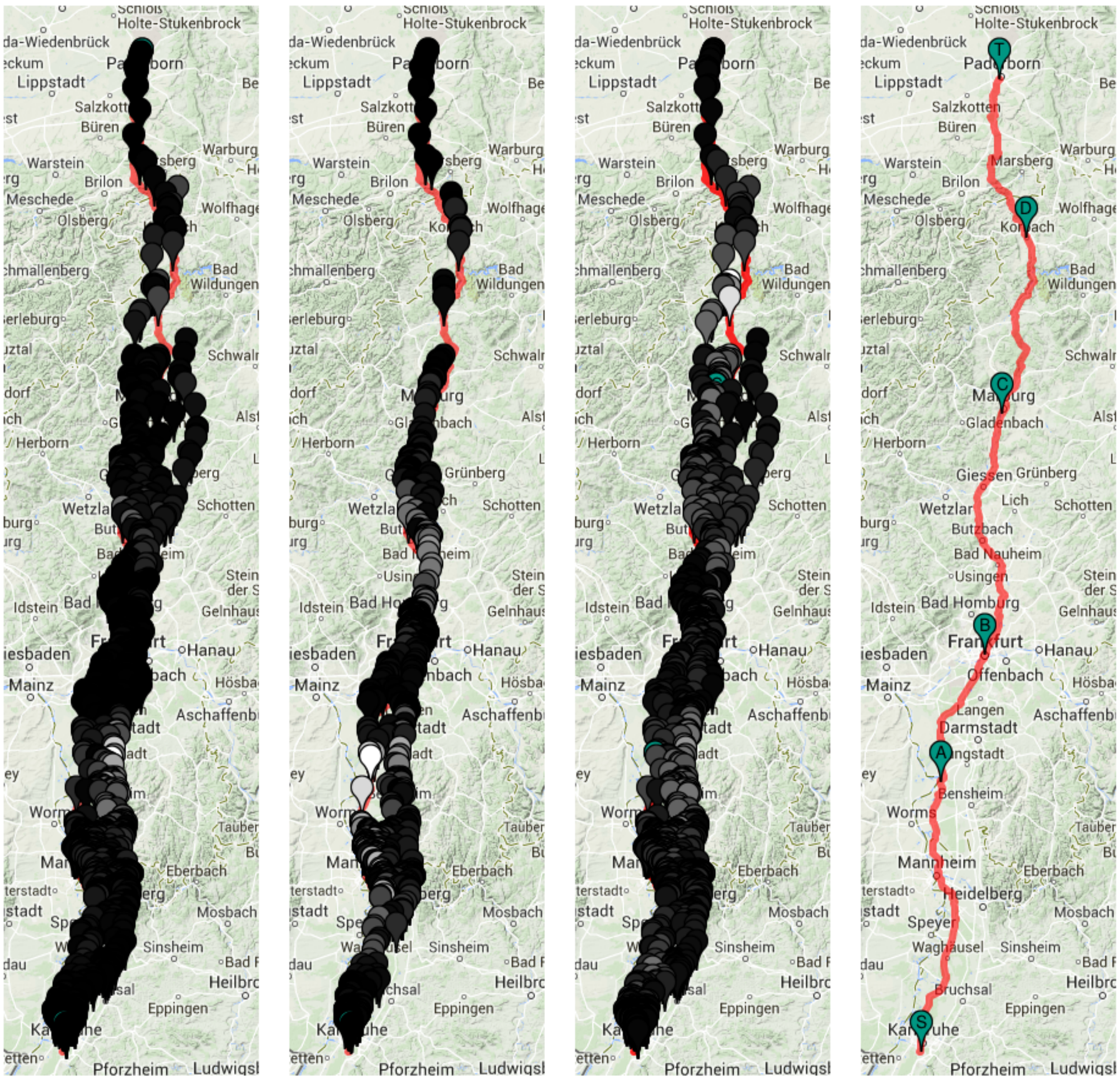

Figure 7.5: Search space of three algorithms (from left to right): exact with potential π_2, exact with potential π_3, heuristic with potential π_2. For this query we used the germany graph with charging station locations from ChargeMap, and regular charging stations providing a charging power of 22 kW. The battery has a capacity of 16 kWh. All three algorithms managed to find exactly the same route, which is shown on the very right. The travel time of the computed route is eight hours and 31 minutes. The route requires four charging stops which are marked by the letters A to D. The total charging time of all four stops is three hours and 23 minutes. The first algorithm (exact, π_2) had a query time of 9.76 ms, settled 1 217 different vertices, and settled 162 426 labels. The second algorithm (exact, π_3) had a query time of 2.37 ms, settled 810 different vertices, and settled 35 778 labels. The third algorithm (heuristic, π_2) had a query time of 0.09 ms, settled 1 217 different vertices, and settled 8 018 labels.

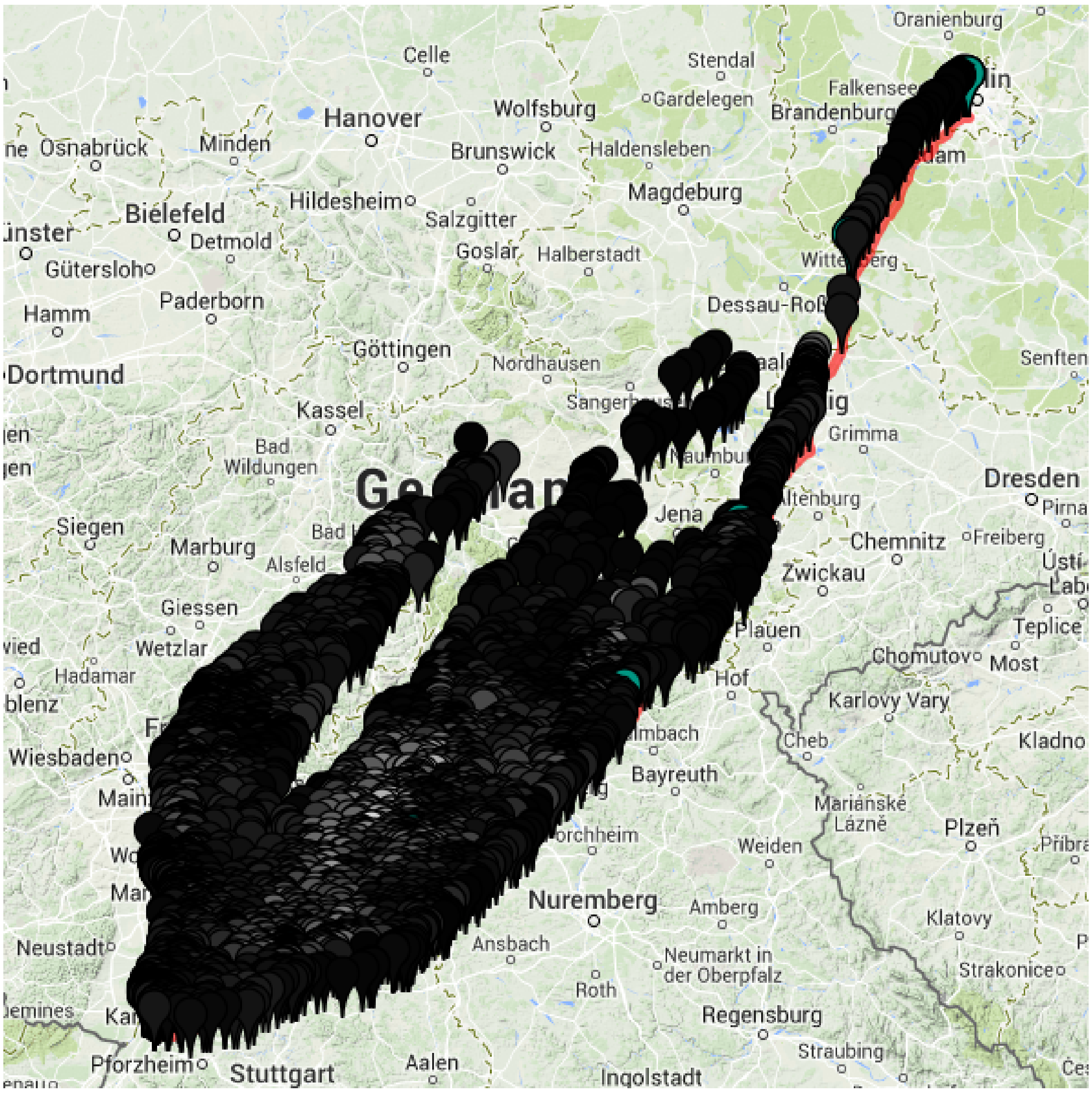

Figure 7.6: The search space of the heuristic algorithm using π_2 as potential for a query from Karlsruhe to Berlin. The query was performed on the germany graph with charging station locations from ChargeMap and regular charging stations with a charging power of 22 kW. Th electric vehicle had a battery capacity of 16 kWh. The algorithm had a query time of 245 ms, settled 3 735 vertices (which are marked on the map), and settled 47 250 labels. The computed route has a travel time of 15 hours, 56 minutes, and 44 seconds. For comparison, the optimal route for this query has a travel time of 15 hours, 55 minutes, and 30 seconds, this is a difference of only 74 seconds. Particularly interesting is the shape of the search space. The search space naturally splits into branches. The reason for this is, that the charging stations are only located along some routes. The query algorithm then explores corridors, which enclose these routes.

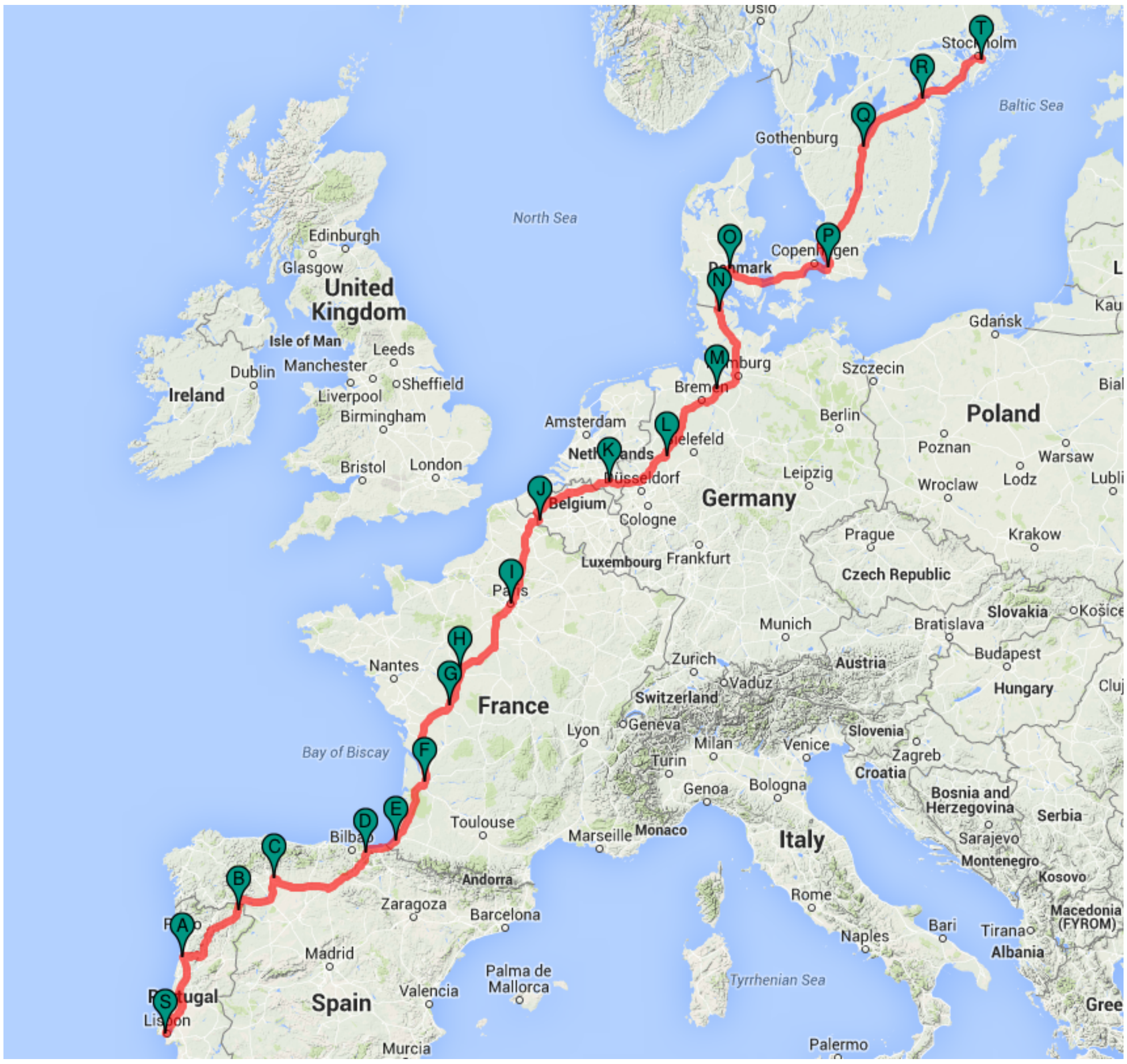

Figure 7.7: Probably one of the longest journeys possible in our graphs. A route from Lisbon (Portugal) to Stockholm (Sweden) using a 60 kWh battery. For this query we used the europe graph with charging station locations from ChargeMap. About 90% of the charging stations are Regular charging stations offering a power of 33 kW (such charging stations are quite common today), the remaining 10% of the charging stations are battery swapping stations (which might be established in the not to far future). Computing the route took us five minutes and four seconds using the heuristic algorithm with π_3 as potential. The computed travel time is one day, twelve hours and 42 minutes. The route requires 18 recharging stops, which are marked with the letters A to R. All of the recharging stops actually use battery swapping stations. This either means that the algorithm by luck encounters swapping stations every time they are needed, or, that the swapping stations outclass regular station is such a way that it is always worth to make a detour in order to prevent regular station from being used.

8. Conclusion

We conclude this thesis with a short summary of the results obtained in this work. Afterwards, we give an outlook on future work emerging from the results and open problems encountered during this thesis.

8.1 Summary

In this thesis we developed a fast algorithm solving the Electric Vehicle Route Planning With Recharging problem. We gave a detailed problem definition and showed NP-hardness in Chapter 3. The problem incorporates both, battery constraints and the presence of various types of charging stations. Furthermore it is allowed to interrupt a charging process at any given time.

We introduced a model for the recharging process, based on piecewise linear and concave functions. These functions are easy to handle in algorithms and provide great flexibility for modeling all kinds of charging stations. Thus we were able, to model different types of charging stations e.g. regular charging stations, super chargers, and battery swapping stations.

Based on the design of the charging functions, we developed our charging function propagating algorithm. This algorithm allows us for the first time to compute a feasible path, minimizing the travel time between two vertices. The algorithm has an exponential worst case complexity, however, it is still capable of computing the optimal solution for small graphs in about one second.

Next, we focused on adapting known speedup techniques to our scenario. We adapted the CH preprocessing as well as A*-search to speedup our base algorithm. For the A* search, we proposed several techniques of computing a potential function which is used to direct the search towards the target. These potentials require different computational effort while they yield different speedups. Using both techniques together, we could achieve a speedup of at least three orders of magnitude, which makes our algorithm applicable for medium-sized graphs.

In order to achieve even faster query times, and to be able to compute feasible paths even for continent-sized graphs, we developed heuristic improvements for our algorithm. Using the heuristics we could achieve an additional speedup of about two orders of magnitude, while the evaluation of the algorithm showed that the quality of heuristic solutions is very high: In most cases the travel time of the heuristic solutions is only some seconds and at most a few minutes longer, than the travel time of the exact solution. This corresponds to an error of 1% or less.

Finally, we evaluated all of our algorithms and heuristics in great detail. We analyzed their performance for various combinations of battery capacity and available charging stations, and on several different graphs. Our experiments show that, on medium-sized graphs, our query times are comparable to the ones of previous solutions. However, our algorithm can handle various types of charging stations, while previous works focused mostly on battery swapping stations. Also, for the first time, we are able to find feasible and fast solutions for continental graphs in only a few minutes using our heuristic algorithm.

8.2 Future Work

While we developed several algorithms throughout this thesis, which are applicable even for large graphs, there is still potential for further improvements.

First of all, further improvements of the query time could be possible due to better vertex potentials. In our algorithm the time needed to compute the potential dominates the overall query time for short range queries. Here it might be possible to use a more efficient potential especially for short range queries, so that such queries can also benefit from the A* speedup technique. Further research on potential functions could also address hyperbolic functions (see Section 5.1.5. If an efficient way of merging these functions is found, they might yield better potential functions than the ones we currently use.

Further research could also address the possibility of saving energy by changing the driving speed on some streets. This idea was already addressed in [BDHS$^+$14]. Combining this approach with the availability of charging stations could yield a more realistically modeled problem than the one we considered in this thesis

Finally, some of the examples shown Sections 7.3 suggest that our algorithm naturally tends to explore several independent paths towards the target, when using our potential functions. It is often desirable to find good alternative routes, so that the user can decide which of them he prefers. Finding such alternative routes is rather difficult when solving the single criterion shortest path problem. Thus it might be interesting to address the problem of finding alternative routes for electric vehicles if charging stations are available.

Acknowledgment

We would like to thank Martin Uhrig, who kindly provided us data sets from his work [UL13], containing detailed charging times for various combinations of battery capacity, charging power, initial SoC, and final SoC. We also thank Sabine Storandt for providing us with the graph and energy consumption model used in her work [Sto12b].

Bibliography

[BDG$^+$14] Hannah Bast, Daniel Delling, Andrew V. Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route Planning in Transportation Networks. Technical Report MSR-TR-2014-4, 2014.

[BDHS$^+$14] Moritz Baum, Julian Dibbelt, Lorenz Hübschle-Schneider, Thomas Pajor, and Dorothea Wagner. Speed-Consumption Tradeoff for Electric Vehicle Route Planning. In *14th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*, pages 138–151, 2014.

[BDPW13] Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. Energy-Optimal Routes for Electric Vehicles. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 54–63. ACM, 2013.

[BDS$^+$08] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-directed Speed-up Techniques for Dijkstra's Algorithm. *Journal of Experimental Algorithmics*, 15:303–318, 208.

[BFSS07] Holger Bast, Stefan Funke, Peter Sanders, and Dominik Schultes. Fast Routing in Road Networks with Transit Nodes. *Science*, 316(5824):566–566, 2007.

[Dij59] Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[DW09] Daniel Delling and Dorothea Wagner. Pareto Paths with SHARC. In *Experimental Algorithms*, pages 125–136. Springer, 2009.

[EFS11] Jochen Eisner, Stefan Funke, and Sabine Storandt. Optimal Route Planning for Electric Vehicles in Large Networks. In *AAAI*, 2011.

[FT87] Michael L. Fredman and Robert E. Tarjan. Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[GH05] Andrew V. Goldberg and Chris Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '05, pages 156–165. Society for Industrial and Applied Mathematics, 2005.

[GJ90] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., 1990.

[GP14] Michael T. Goodrich and Paweł Pszona. Two-Phase Bicriterion Search for Finding Fast and Efficient Electric Vehicle Routes. *arXiv preprint arXiv:1409.3192*, 2014.

[Gra72] Ronald L. Graham. An Efficient Algorithm for Determining the Convex Hull of a Finite Planar Set. *Information Processing Letters*, 1(4):132–133, 1972.

[GSSD08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *Experimental Algorithms*, pages 319–333. Springer, 2008.

[HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968.

[HRZL09] Stefan Hausberger, Martin Rexeis, Michael Zallinger, and Raphael Luz. Emission Factors from the Model PHEM for the HBEFA Version 3. *University of Technology, Graz, Report Nr. I-20/2009 Haus-Em*, 33(08):679, 2009.

[HS13] Lorenz Hübschle-Schneider. Speed–Consumption Trade-Off for Electric Vehicle Routing. Bachelor thesis, Karlsruhe Institute of Technology, 2013.

[HZ80] Gabriel Y. Handler and Israel Zang. A Dual Algorithm for the Constrained Shortest Path Problem. *Networks*, 10(4):293–309, 1980.

[Joh73] Donald B. Johnson. A Note on Dijkstra's Shortest Path Algorithm. *Journal of the ACM*, 20(3):385–388, 1973.

[JW06] Andreas Jossen and Wolfgang Weydanz. *Moderne Akkumulatoren richtig einsetzen*. Inge Reichardt Verlag, 2006.

[KMS06] Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast Point-to-Point Shortest Path Computations with Arc-flags. *9th DIMACS Implementation Challenge*, 2006.

[LWL14] Chensheng Liu, Jing Wu, and Chengnian Long. Joint Charging and Routing Optimization for Electric Vehicle Navigation Systems. 2014.

[Mar84] Ernesto Q. V. Martins. On a Multicriteria Shortest Path Problem. *European Journal of Operational Research*, 16(2):236–245, 1984.

[MHW01] Matthias Müller-Hannemann and Karsten Weihe. Pareto Shortest Paths is Often Feasible in Practice. In *Algorithm Engineering*, pages 185–197. Springer, 2001.

[SBW12] Olivia J. Smith, Natashia Boland, and Hamish Waterer. Solving Shortest Path Problems with a Weight Constraint and Replenishment Arcs. *Computers & Operations Research*, 39(5):964–984, 2012.

[SDK14] Timothy M. Sweda, Irina S. Dolinskaya, and Diego Klabjan. Adaptive Routing and Recharging Policies for Electric Vehicles. 2014.

[SF12] Sabine Storandt and Stefan Funke. Cruising with a Battery-Powered Vehicle and Not Getting Stranded. In *AAAI*, volume 3, page 46, 2012.

[SI91] Bradley S. Stewart and Chelsea C. White III. Multiobjective A*. *Journal of the ACM (JACM)*, 38(4):775–814, 1991.

[SLAH11] Martin Sachenbacher, Martin Leucker, Andreas Artmeier, and Julian Haselmayr. Efficient Energy-Optimal Routing for Electric Vehicles. In *AAAI*, 2011.

[SM13] Peter Sanders and Lawrence Mandow. Parallel label-Setting Multi-Objective Shortest Path Search. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 215–224. IEEE, 2013.

[Sto12a] Sabine Storandt. *Algorithms for Vehicle Navigation*. PhD thesis, Universität Stuttgart, 2012.

[Sto12b] Sabine Storandt. Quick and Energy-Efficient Routes: Computing Constrained Shortest Paths for Electric Vehicles. In *Proceedings of the 5th ACM SIGSPATIAL International Workshop on Computational Transportation Science*, pages 20–25. ACM, 2012.

[Sto12c] Sabine Storandt. Route Planning for Bicycles - Exact Constrained Shortest Paths Made Practical via Contraction Hierarchy. In *ICAPS*, volume 4, pages 234–242, 2012.

[TC92] Chi Tung Tung and Kim Lin Chew. A Multicriteria Pareto-Optimal Path Algorithm. *European Journal of Operational Research*, 62(2):203–209, 1992.

[UL13] Martin Uhrig and Thomas Leibfried. Ausbaubedarf von Parkhäusern zum Laden von Elektrofahrzeugen. In *ETG-Fachbericht-Internationaler ETG-Kongress 2013–Energieversorgung auf dem Weg nach 2050*. VDE VERLAG GmbH, 2013.

Appendix

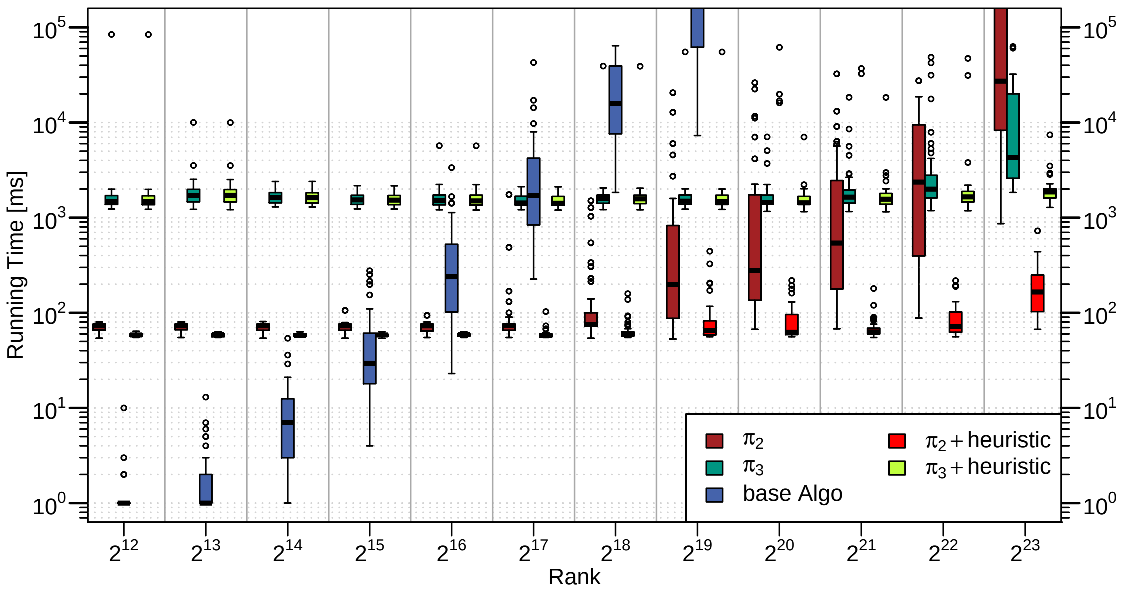

Figure .1: Dijkstra rank plot evaluating our algorithms on the germany graph, for a battery capacity of 16 kWh.

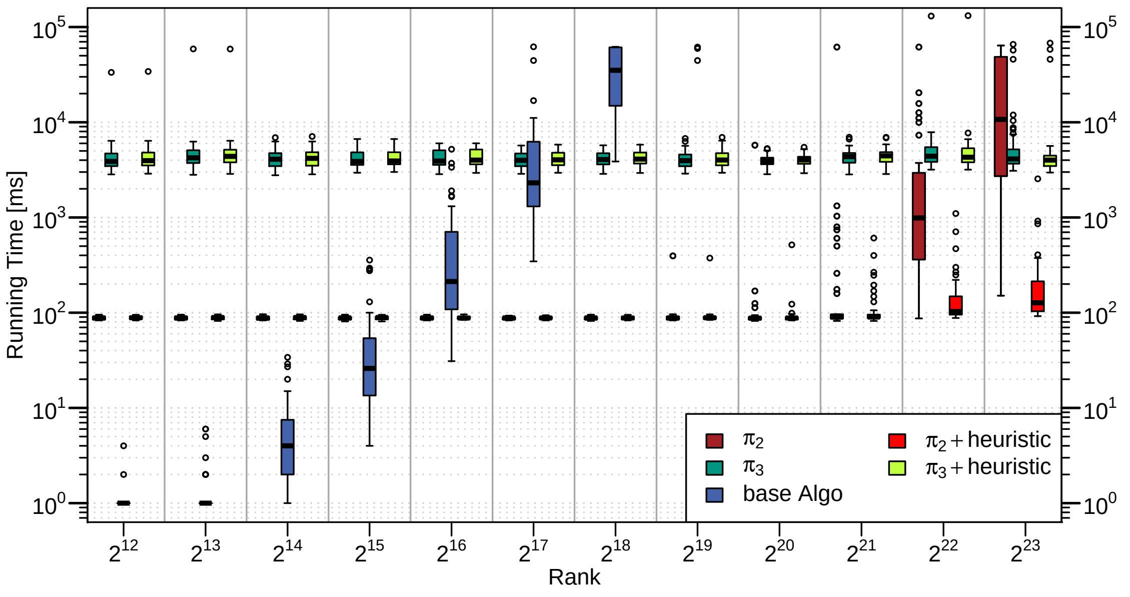

Figure .2: Dijkstra rank plot evaluating our algorithms on the germany graph, for a battery capacity of 60 kWh.

Table .1: Charging times for a 16 kWh battery at a charging station offering a power of 11 kW. The data is kindly provided by Martin Uhrig.

Initial SoC [%]	Final SoC [%]																			
	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100
0	262	525	787	1 061	1 323	1 595	1 857	2 119	2 381	2 642	2 904	3 166	3 428	3 690	3 951	4 233	4 525	4 937	5 543	7 339
5	0	262	525	787	1 060	1 322	1 584	1 846	2 108	2 369	2 631	2 893	3 155	3 417	3 678	3 940	4 242	4 654	5 260	7 056
10	0	0	262	524	786	1 053	1 315	1 577	1 839	2 100	2 362	2 624	2 886	3 148	3 409	3 671	3 973	4 385	4 991	6 787
15	0	0	0	263	525	787	1 049	1 311	1 573	1 834	2 096	2 358	2 620	2 882	3 143	3 405	3 707	4 119	4 725	6 521
20	0	0	0	0	262	524	786	1 048	1 310	1 571	1 833	2 095	2 357	2 619	2 880	3 142	3 444	3 856	4 462	6 258
25	0	0	0	0	0	262	524	786	1 048	1 310	1 571	1 833	2 095	2 357	2 619	2 880	3 182	3 594	4 200	5 996
30	0	0	0	0	0	0	262	524	786	1 048	1 310	1 571	1 833	2 095	2 357	2 619	2 921	3 333	3 939	5 735
35	0	0	0	0	0	0	0	262	524	786	1 048	1 310	1 571	1 833	2 095	2 357	2 659	3 071	3 677	5 473
40	0	0	0	0	0	0	0	0	262	524	786	1 048	1 310	1 571	1 833	2 095	2 397	2 809	3 415	5 211
45	0	0	0	0	0	0	0	0	0	262	524	786	1 048	1 310	1 571	1 833	2 135	2 547	3 153	4 949
50	0	0	0	0	0	0	0	0	0	0	262	524	786	1 048	1 310	1 571	1 873	2 285	2 891	4 687
55	0	0	0	0	0	0	0	0	0	0	0	262	524	786	1 048	1 310	1 612	2 024	2 630	4 426
60	0	0	0	0	0	0	0	0	0	0	0	0	262	524	786	1 049	1 350	1 762	2 368	4 164
65	0	0	0	0	0	0	0	0	0	0	0	0	0	262	524	786	1 088	1 500	2 106	3 902
70	0	0	0	0	0	0	0	0	0	0	0	0	0	0	262	525	826	1 238	1 844	3 640
75	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	262	564	976	1 582	3 378
80	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	302	714	13 20	3 116
85	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	413	1 019	2 815
90	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	607	2 403
95	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1 797