Karlsruhe Institute of Technology

Energy-Optimal Routes for Electric Vehicles with Charging Stops

Bachelor Thesis of

Jonas Sauer

At the Department of Informatics
Institute of Theoretical Computer Science

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. Peter Sanders
Advisors:	Moritz Baum, M.Sc.
	Tobias Zündorf, M.Sc.

Time Period: 15th June 2015 – 14th October 2015

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, 13th October 2015

Abstract

The importance of electric vehicles has increased steadily in recent years. As battery capacities increase and charging stations for electric vehicles become more widespread, long-distance routes that require charging stops are starting to become feasible. While route planning problems for electric vehicles are well-studied, little research has focused on incorporating charging stops. In this thesis we extend previous work on the subject to develop two algorithms for finding energy-optimal routes while taking several types of charging stations into account. Our first algorithm is based on a multi-objective shortest path search which uses Pareto sets. While multi-objective shortest path search algorithms generally have an exponential worst-case complexity, we introduce a class of graphs for which we prove a polynomial complexity of our algorithm. The second algorithm uses a precalculated graph that contains the shortest paths between all charging stations to speed up the queries. We evaluate our algorithms on real-world road network data, observing that both algorithms can answer queries on the road network of Germany in a matter of seconds.

Deutsche Zusammenfassung

Die Bedeutung von Elektrofahrzeugen hat in den letzten Jahren stetig zugenommen. Dank steigender Batteriekapazitäten und zunehmender Verbreitung von Ladestationen für Elektrofahrzeuge werden Langstreckenrouten mit Ladestopps allmählich praktikabler. Während Routenplanungsprobleme für Elektrofahrzeuge bereits ausführlich untersucht wurden, hat sich nur wenig Forschung mit der Berücksichtigung von Ladestationen beschäftigt. In dieser Arbeit erweitern wir frühere Arbeiten zu diesem Thema und entwickeln zwei Algorithmen, die Energie-optimale Routen finden und dabei Ladestopps erlauben. Unser erster Algorithmus basiert auf einer multikriteriellen Kürzeste-Wege-Suche, die Pareto-Mengen verwendet. Obwohl multikriterielle Kürzeste-Wege-Algorithmen im Allgemeinen eine exponentielle Worst-Case-Komplexität haben, führen wir eine Klasse von Graphen ein, für die wir eine polynomielle Laufzeit unseres Algorithmus zeigen. Der zweite Algorithmus verwendet einen vorberechneten Graph, der die kürzesten Wege zwischen alle Ladestationen enthält, um die Anfragen zu beschleunigen. Wir werten unsere Algorithmen für echte Straßennetzwerkdaten aus und beobachten für beide Algorithmen Anfragezeiten im Sekundenbereich auf dem Straßennetzwerk Deutschlands.

Contents

1. Introduction

In recent years, the importance of electric vehicles as a fossil fuel independent alternative to conventional vehicles has increased steadily. However, their limited battery capacity, which leads to comparatively small cruising ranges, still presents a major obstacle towards a more widespread use. While the number of charging stations for electric vehicles is increasing, they are still fairly rare compared to conventional gas stations. Furthermore, the process of recharging the battery is time-consuming and may take several hours. This makes it especially important to plan long-distance routes with charging stops beforehand, presenting a new challenge for routing algorithms.

The constraints imposed by an electric vehicle's limited battery capacity require special algorithmic handling. In particular, route planning algorithms must ensure that the vehicle never runs out of energy. Furthermore, electric vehicles can recharge their battery when driving downhill; this effect is called *recuperation*. However, recuperation is only possible as long as the battery does not exceed its maximum capacity. Therefore, routing algorithms for electric vehicles must keep track of the battery's *state of charge* to ensure that it always remains within these constraints.

Previous research on the topic of electric vehicle route planning has focused mostly on finding time- or energy-optimal routes while taking battery constraints into account. Research on incorporating charging stations has been limited so far, mainly because small battery capacities and long charging times make long-distance routes impractical. However, with the advent of more powerful charging stations and vehicles with larger batteries, charging stops are becoming increasingly feasible.

For the problem of finding energy-optimal routes, only battery swapping stations (BSS) have been considered as a method of recharging. These are stations where the entire battery is replaced with a new, fully charged battery. However, battery swapping stations are currently not widely available. So far, route planning algorithms that take commonly available types of charging stations into account exist only for the time-optimal electric vehicle routing problem. In this thesis, we adapt existing approaches to develop algorithms that find energy-optimal routes and can handle a variety of charging station types.

1.1 Related Work

The basis for many route planning algorithms is Dijkstra's algorithm [Dij59], which finds the shortest path between two points in a road network according to a single, static metric. In the context of electric vehicles, this metric is usually either driving time or energy consumption. For metrics without negative values, such as driving time, Dijkstra's algorithm can be implemented with a worst-case complexity of $\mathcal{O}(n \log n + m)$, where m is the number of edges in the road network graph (which represent road segments) and n is the number of vertices (which represent end points and intersections of road segments). Due to the capability of electric vehicles to recuperate energy, the energy consumption on some road segments may be negative. In this case, the running time of Dijkstra's algorithm may become exponential in the size of the road network.

The main challenge when developing route planning algorithms for electric vehicles is to take into account the limitations imposed by the vehicle's limited battery capacity: Once the battery's entire energy has been consumed, the vehicle becomes stranded and cannot drive any further. Consuming further energy when the battery is empty, which we call *undercharging*, is impossible. Similarly, while the vehicle can recuperate energy when driving downhill, it cannot do so if the battery is full. We call this situation *overcharging*. Both undercharging and overcharging have to be avoided by an electric vehicle-aware routing algorithm.

The energy-optimal route planning problem for electric vehicles, where the objective is to minimize the overall energy consumption, was first introduced by Artmeier et al. [AHLS10]. Eisner et al. [EFS11] modified Dijkstra's algorithm to solve the problem by replacing the static edge costs with functions that take the battery constraints into account. To remove negative edge costs, they adapt a technique called *potential shifting*, which was introduced by Johnson [Joh77]. Furthermore, they adapt Contraction Hierarchies (CH), a speedup technique introduced by Geisberger et al. [GSSD08], to obtain running times of less than a second even on large road network graphs with several million road segments.

Baum et al. [BDPW13] improved on this by adapting a speedup technique called Customizable Route Planning (CRP), first developed by Delling [DGPW11]. This yields an average running time of less than 5 milliseconds even on the road network of Europe, making the algorithm feasible for interactive applications. As part of this approach, they also developed a method for answering *profile queries*: Whereas regular queries specify an initial SoC for the vehicle at the start of the route, a profile query asks for the shortest path for every possible initial SoC value.

Little research has been done so far on incorporating charging stations into route planning for electric vehicles. For the energy-optimal version of the problem, Storandt and Funke [SF12] developed an approach that only considers battery swapping stations as a possible method of recharging. Their algorithm makes use of a precalculated auxiliary graph that contains information about the shortest paths between all charging stations. To answer a query, it then computes the set of all charging stations that can be reached from the source vertex without recharging, as well as the set of all charging stations from which the target vertex can be reached without recharging. The auxiliary graph is then used to calculate the remainder of the path between these two sets of charging station.

The first algorithm that allows for various types of charging stations was developed by Zündorf [Zün14] for the time-optimal electric vehicle routing problem. Specifically, this algorithm considers three basic types of charging stations: In addition to regular charging stations, where the battery can be recharged to any desired SoC, and the previously mentioned battery swapping stations, it also considers superchargers. These are charging stations with a high amount of charging power, allowing the battery to be recharged much faster than at a regular charging station. However, due to technical limitations, the achievable SoC is limited to 80% of the maximum battery capacity. Zündorf uses piecewise linear functions to model the charging process, developing an algorithm that minimizes the overall travel time, which includes both driving time and charging time.

1.2 Contribution

In this thesis we incorporate charging stations into the energy-optimal shortest path problem for electric vehicles and develop two algorithms for solving this problem. As in [Zün14], we consider regular charging stations, superchargers and battery swapping stations as the three basic types of charging stations. While we do not consider the overall charging time, we minimize the number of charging stops to ensure that the routes are realistically usable.

The first algorithm we develop is based on the approach from [Zün14], using piecewise linear functions to model the charging process. We show that in the energy-optimal case, these functions can be simplified and represented entirely by static values. We then analyze the complexity of the developed algorithm. Whereas the original time-optimal version of the problem is $\mathcal{NP}$-complete, we introduce a class of graphs on which our algorithm can be proven to have a polynomial worst-case complexity.

For the second algorithm, we adapt the approach from [SF12], using an auxiliary graph to precalculate the shortest paths between charging stations. By using profile queries, we extend this approach to allow for other charging station types besides battery swapping stations.

Finally, we analyze the performance of both algorithms on real-world road network data and compare the results.

1.3 Outline

The remainder of this thesis is structured as follows:

Chapter 2. In the preliminaries chapter, we lay the foundation for the thesis, introducing basic concepts and notation. We present algorithms for solving the shortest path problem and the energy-optimal shortest path problem for electric vehicles, which serve as a basis for our algorithms.

Chapter 3. A precise statement of the problem solved by our algorithms is given in this chapter.

Chapter 4. In this chapter, we develop our first algorithm. We start by adapting the approach of using piecewise linear functions to model the charging process, which we then show can be simplified in our case. After stating our basic approach for the algorithm, we discuss several methods for improving its performance.

Chapter 5. We continue by analyzing the complexity of the algorithm developed in the previous chapter, introducing a class of graphs for which we can show a polynomial worst-case complexity of our algorithm.

Chapter 6. In this chapter, we developed a second algorithm for solving our problem, which uses a precalculated auxiliary graph to speed up the queries. After stating the basic approach, we explain how it can be adapted to consider different types of charging stations.

Chapter 7. In the evaluation chapter, we conduct experiments with both algorithms on real-word road network data, evaluating their performance and comparing the results.

Chapter 8. Finally, we conclude the thesis with a summary of the achieved results and an outlook on possible future work.

2. Preliminaries

In this chapter we introduce basic concepts and algorithms that lay the foundation for the remaining thesis.

2.1 Graph Theory

Graphs. A *graph* $G = (V, E)$ is a tuple consisting of a set V of *vertices* and a set $E \subseteq V \times V$ of *edges*. We only consider *directed* graphs, which means that an edge $e = (u, v) \in E$ is always directed from u to v. We call u the *tail* and v the *head* of e. The number of vertices in a graph is denoted by $n \coloneqq |V|$ and the number of edges is denoted by $m \coloneqq |E|$. A graph $G' = (V', E')$ is called a *subgraph* of another graph $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$ hold.

For a vertex $u \in V$, the number of edges whose tail is u is called the *out-degree* of u; it is defined as $\deg_{\text{out}}(u) \coloneqq |\{(u, v) \in E \mid v \in V\}|$. Similarly, the number of edges whose head is u is called the *in-degree* of u; it is defined as $\deg_{\text{in}}(u) \coloneqq |\{(v, u) \in E \mid v \in V\}|$. The number of all edges that are incident to u is called the *degree* of u; naturally, it is the sum of the out-degree and the in-degree: $\deg(u) \coloneqq \deg_{\text{out}}(u) + \deg_{\text{in}}(u)$.

In order to find energy-optimal routes in a road network, we need information about the energy consumption on the roads, which are represented by edges. To model this information, we use a *scalar edge weight function* $\omega\colon E \to \mathbb{R}$, which assigns a value to every edge in the graph. In our case this value is the energy consumption, so we define the energy consumption edge weight $\text{cons}\colon E \to \mathbb{R}$, which assigns to every edge the energy that is consumed when traversing it.

Paths. A *path* P in a graph G is a sequence of vertices $(v_1, v_2, \ldots, v_k)$ in V such that an edge $(v_{i-1}, v_i) \in E$ exists for $1 < i \leq k$. The number of vertices contained in the path is denoted by $|P| = k$. A path where $v_1 = v_k$ is called a *cycle*. We call a vertex $v \in V$ *reachable* from another vertex $u \in V$ if there exists a path $P = (u, \ldots, v)$ in G that starts at u and ends at v. We call such a path a u-v-path. If u is reachable from v as well as v from u, we call u and v *connected*.

A path $Q = (u_1, u_2, \ldots, u_l)$ is called a *subpath* of another path $P = (v_1, v_2, \ldots, v_k)$ (denoted $Q \subset P$) if $l \leq k$ and there exists an $i \in [0, k - l]$ such that for all $0 < j \leq l$ the condition $v_{i+j} = u_j$ holds. The subpath of P that contains only the first i vertices is called a *prefix* of P and is denoted by $P^i \coloneqq (v_1, v_2, \ldots, v_i)$.

We can extend the definition of edge weights to also cover paths. Given an edge weight function ω on E, the weight (or *length*) of a path $P = (v_1, v_2, \ldots, v_k)$ is defined as the sum of the edge weights in the path: $\omega(P) \coloneqq \sum_{i=2}^{k} \omega((v_{i-1}, v_i))$.

A *tree* $\mathcal{T}_u$ *rooted* at a vertex u is a graph with $m = n - 1$ edges such that there is a unique path from u to every other vertex v. The root vertex u is the only vertex in $\mathcal{T}_u$ whose in-degree is zero, all other vertices have an in-degree of one. A vertex v in $\mathcal{T}_u$ is called a *leaf* if its out-degree is zero.

Shortest Paths. Given two vertices $u, v \in V$ and an edge weight function ω, the *shortest* u-v-*path* P is the one with minimal weight $\omega(P)$ among all u-v-paths in G. The *distance* between u and v is formally defined as $\text{dist}_\omega(u, v) \coloneqq \inf_{P=(u,\ldots,v)} \omega(P)$. This is equivalent to the length of the shortest u-v-path in all but two cases: If u and v are not connected, there is no shortest u-v-path and $\text{dist}_\omega(u, v)$ is thus ∞. If there exists a u-v-path that contains a negative cycles, $\text{dist}_\omega(u, v)$ is $-\infty$. However, road networks can never contain cycles with a negative energy consumption, since this would mean a vehicle could recharge the battery infinitely by continuing to drive around the cycle. Therefore, we know that in our case, $\text{dist}_\omega(u, v) \in \mathbb{R} \cup \{\infty\}$ for all vertices $u, v \in V$.

A *shortest path tree* $\mathcal{T}_u$ *rooted* at a vertex u is a subgraph of G that contains all vertices that are reachable from u and is a tree rooted at u where the unique path from u to every other vertex v is a shortest path in G.

2.2 Functions

A *function* $f\colon X \to Y$ is a relation between two sets X and Y that assigns to each $x \in X$ a value $f(x) \in Y$. We call x an *argument* of f and $f(x)$ the *value* of f for x. The tuple $(x, f(x))$ is called a *point* of f.

A *piecewise linear function* $f\colon X \to Y$ is a function that can be decomposed into a sequence of continuous intervals $(I_1 = [x_1, x_2), I_2 = [x_2, x_3), \ldots, I_k = [x_k, \infty))$ so that f restricted to each interval I_i $(1 \le i \le k)$ is linear. We call the endpoints $(x_i, f(x_i))$ of these intervals the *supporting points* of f. Inside each interval I_i, the derivative of f is well-defined, except at the supporting points themselves. We call the value of the derivative of f in I_i the *slope* of f in I_i.

Given a sequence of supporting points and slopes $[(x_1, y_1, s_1), (x_2, y_2, s_2), \ldots, (x_k, y_k, s_k)]$, we can define the corresponding piecewise linear function f that goes through these supporting points and has these slopes:

$$f(x) \coloneqq \begin{cases} y_1 + s_1(x - x_1) & \text{if } x_1 \le x < x_2 \\ y_2 + s_2(x - x_2) & \text{if } x_2 \le x < x_3 \\ \vdots & \\ y_k + s_k(x - x_k) & \text{if } x_k \le x. \end{cases}$$

2.3 The Shortest Path Problem

In this section, we introduce problems related to the computation of shortest paths and an algorithm for solving them.

Definition 2.1. *We are given a graph $G = (V, E)$ and an edge weight ω. We define four basic shortest path problems:*

Algorithm 2.1: DIJKSTRA'S ALGORITHM

```
Input: Graph $G = (V, E)$, edge weight function $\omega\colon E \to \mathbb{R}$, source vertex $s$
Data: Priority queue Q
Output: Distances from $s$ given by dist[·], shortest-path tree of $s$ given by parent[·]

// Initialization
1  forall $v \in V$ do
2      dist[$v$] $\leftarrow \infty$
3      parent[$v$] $\leftarrow \perp$
4  dist[$s$] $\leftarrow 0$
5  Q.INSERT($s$, dist[$s$])
// Main loop
6  while not Q.isEmpty() do
7      $u \leftarrow$ Q.deleteMin()
8      forall $(u, v) \in E$ do
9          if dist[$u$] $+ \omega((u, v)) <$ dist[$v$] 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[$v$])
```

- *The* SINGLE-PAIR SHORTEST PATH (SPSP) *Problem: Given source and target vertices $s, t \in V$, we ask for the shortest s-t-path.*
- *The* SINGLE-SOURCE SHORTEST PATH (SSSP) *Problem: Given a source vertex $s \in V$, we ask for the shortest path from s to every other vertex in V.*
- *The* SINGLE-DESTINATION SHORTEST PATH (SDSP) *Problem: Given a target vertex $t \in V$, we ask for the shortest path from every other vertex in V to t.*
- *The* ALL-PAIRS SHORTEST PATH (APSP) *Problem: We ask for the shortest path between every pair of vertices in V.*

The goal of this thesis is to solve an extension of the SPSP problem that takes battery constraints into account and allows for charging stops. As part of the algorithms we will develop, we will also encounter the SSSP and SDSP problems. All three of these problems are solved by Dijkstra's algorithm.

2.3.1 Dijkstra's Algorithm

Dijkstra's algorithm is a well-known algorithm for solving the single-source shortest path problem (and, by extension, also the single-pair and single-destination shortest path problems). It was first published by Edsger W. Dijkstra in 1959 [Dij59]. Provided that all edge weights in the graph are non-negative, Dijkstra's algorithm has a polynomial running time in the size of the graph. If this is not the case, the running time may be exponential, but the algorithm will still terminate as long as the graph contains no negative cycles. The pseudocode for Dijkstra's algorithm is provided in Algorithm 2.1.

The algorithm takes a graph $G = (V, E)$, an edge weight function $\omega\colon E \to \mathbb{R}$ and a source vertex s as input. It outputs two arrays, dist[·] and parent[·]. After the algorithm has terminated, $\mathsf{dist}[v]$ holds the distance $\mathrm{dist}_\omega(s,v)$ from s to v for every vertex $v \in V$. The $\mathsf{parent}[\cdot]$ array is an implicit representation of the shortest path tree rooted at s: For every vertex $v \in V$ that is reachable from s, with the shortest s-v-path being $\mathrm{SP}_\omega(s,v) = (s = v_1, v_2, \ldots, v_{k-1}, v_k = v)$, $\mathsf{parent}[v]$ points to the vertex v_{k-1} that precedes v in P. By following the pointers in $\mathsf{parent}[\cdot]$ from v to s, the shortest path P can be reconstructed with the rule $v_{i-1} = \mathsf{parent}[v_i]$ for all $1 < i \leq k$.

As an auxiliary data structure, the algorithm uses a priority queue Q. This queue contains key-value pairs, where the value represents a vertex $v \in V$ and the key represents the tentative distance from s given by $\mathsf{dist}[v]$. Q must provide the following operations:

- `insert(`v`,`d`)` inserts the vertex v into the queue and sets its key to d. A necessary precondition for this operation is that v is not already contained in the queue.
- `isEmpty()` returns a boolean value, which is `true` if and only if the queue contains no elements.
- `deleteMin()` finds the vertex v with the smallest key of all elements in the queue, deletes it from the queue and returns it.
- `contains(`v`)` returns a boolean value, which is `true` if and only if the queue contains v as an element.
- `decreaseKey(`v`,`d`)` finds the element whose value is v and sets its key to d. Necessary preconditions for this operation are that v is contained in the queue and that the current key of v is greater than or equal to d.

The idea of Dijkstra's algorithm is to expand from s in increasing order of distance. In the first step, the arrays $\mathsf{dist}[\cdot]$ and $\mathsf{parent}[\cdot]$ as well as the priority queue Q are initialized. $\mathsf{dist}[s]$ is set to 0. Since we do not yet know the distances to any other vertices in the graph, we set $\mathsf{dist}[v] = \infty$ for all other vertices $v \in V$. Likewise, since we have not found any paths yet, $\mathsf{parent}[v]$ is initialized with $\perp$ for every vertex $v \in V$. We initialize Q with s as its only element. Once a vertex v has been inserted into Q, we call it *visited*; once it has been removed from Q, we call it *settled*.

Once the initialization has finished, we begin the main loop of the algorithm, which continues until Q is empty. In each step of the main loop, the algorithm extracts the vertex u with the smallest key from Q and settles it. Among all visited but unsettled vertices, u is closest to s. If the graph contains no negative edges, $\mathsf{dist}[u]$ now holds the correct distance from s and the shortest s-u-path can be found via $\mathsf{parent}[u]$. This property is called the *label-setting* property of Dijkstra's algorithm. It means that once a vertex has been settled, it will not be re-inserted into Q.

After settling u, the algorithm scans 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 e is shorter than the current tentative distance $\mathsf{dist}[v]$ of v. If this is the case, we *relax* e, which means that we set $\mathsf{dist}[u] + \omega(e)$ as the new tentative distance of v and u as the new parent of v. We then update Q to reflect the changes to v: If v has already been visited before, it is already contained in Q, so we merely update its key. If v is not contained in Q, we insert it.

Correctness. We give a proof of correctness of Dijkstra's algorithm for the case that the graph contains no negative edges. Correctness in this case means that after the algorithm has terminated, the array $\mathsf{dist}[\cdot]$ holds the correct distances from s, i.e., $\mathsf{dist}[v] = \mathrm{dist}_\omega(s,v)$ for all $v \in V$.

Theorem 2.2. *The following three statements are true for Dijkstra's algorithm:*

1. *Every vertex that is reachable from s gets settled in the course of the algorithm.*
2. *The invariant* $\mathsf{dist}[v] \geq \mathrm{dist}_\omega(s, v)$ *holds throughout the course of the algorithm for all vertices $v \in V$, i.e., the tentative distance of a vertex is never smaller than the actual distance.*
3. *Label-setting property: Once a vertex v is settled,* $\mathsf{dist}[v] = \mathrm{dist}_\omega(s, v)$ *holds.*

Proof. We prove the first claim by contradiction. Assume there is a vertex v that is reachable from s but is never settled by the algorithm. We know that every vertex that is visited (and thus inserted into Q) is also settled at some point, since the algorithm only terminates once Q is empty. Thus, v was also never visited. Since v is reachable from s, there exists a path $P = (s = v_1, v_2, \dots, v_k = v)$ from s to v. s was settled in the first iteration of the main loop, so we know there must be an $i \in [2, k]$ such that v_{i-1} was settled but v_i was not. Once v_{i-1} is settled, the edge $e = (v_{i-1}, v_i)$ is examined. Since v_i was never visited, its tentative distance is ∞. The length of P is obviously smaller than this, so e is relaxed and v is inserted into Q. This is a contradiction to the assumption that v is never settled.

The second claim is true because the tentative distance $\mathsf{dist}[v]$ always corresponds to the path that can be retrieved by following $\mathsf{parent}[v]$. Since $\mathrm{dist}_\omega(s, v)$ corresponds to the length of the shortest s-v-path, and the path corresponding to $\mathsf{dist}[v]$ is thus at least as long, we know that $\mathsf{dist}[v] \geq \mathrm{dist}_\omega(s, v)$.

The third claim is again proven by contradiction. Assume that there is a vertex v for which $\mathsf{dist}[v] > \mathrm{dist}_\omega(s, v)$ holds after being settled. Since v is reachable from s, we know that a shortest s-v-path $\mathrm{SP}_\omega(s, v) = (s = v_1, v_2, \dots, v_k)$ exists. This path cannot be the one that is represented by $\mathsf{parent}[v]$, since otherwise $\mathsf{dist}[v] = \omega(\mathrm{SP}_\omega(s, v)) = \mathrm{dist}_\omega(s, v)$ would hold. Therefore, $\mathrm{SP}_\omega(s, v)$ must contain at least one unsettled vertex. However, this is impossible due to the first claim, because all vertices in $\mathrm{SP}_\omega(s, v)$ are obviously reachable from s. Therefore, we know that $\mathsf{dist}[v] = \mathrm{dist}_\omega(s, v)$ holds. □

From the third statement, it follows that Dijkstra's algorithm computes the correct distance from s to every vertex v.

Complexity. We now analyze the complexity of Dijkstra's algorithm for the case that the graph contains no negative edges. The initialization phase takes $\mathcal{O}(n)$ time, since every vertex is initialized once. Due to the label-setting property, each vertex is visited and settled at most once, so the main loop has at most n iterations. Therefore, `isEmpty`(), `deleteMin`() and `insert`$(\cdot, \cdot)$ are each called at most n times. This also means that each edge is relaxed at most once, so `contains`$(\cdot)$ and `decreaseKey`$(\cdot, \cdot)$ are each called at most m times. Altogether this yields a running time of

$$\mathcal{O}(n(1 + T_{\mathtt{isEmpty}} + T_{\mathtt{deleteMin}} + T_{\mathtt{insert}}) + m(T_{\mathtt{contains}} + T_{\mathtt{decreaseKey}}).$$

The exact running time depends on how the priority queue is implemented. The best known worst-case complexity is $\mathcal{O}(m + n \log n)$, which is achieved with a Fibonacci heap [FT87]. In many practical instances, however, a binary heap is sufficient, which yields a worst-case complexity of $\mathcal{O}((m + n) \log n)$.

Stopping Criterion. The algorithm as described above solves the single-source shortest path problem. If we want to solve the single-pair shortest path problem, i.e., we only want to find the shortest path to a single vertex t, we can terminate the algorithm once we know $\mathrm{dist}_\omega(s, t)$. Due to the label-setting property, this is the case once t is settled.

Therefore, we can insert a line after line 7 in Algorithm 2.1 that terminates the loop if $u = t$ holds. This improves the running time of the algorithm in most cases, but it does not affect the worst-case complexity.

Backwards Search. If we want to solve the single-destination shortest path problem for a target vertex t, the algorithm as described so far is not sufficient, since it assumes the existence of a single source vertex. While running Dijkstra's algorithm once for every vertex $v \in V$ while using v as the source vertex and t as the target vertex would solve the problem, this approach is inefficient. Instead, Dijkstra's algorithm can be modified to search backwards from the target vertex instead of forward from the source vertex, which will yield a backwards shortest path tree that contains the shortest path from every vertex $v \in V$ to t. The backwards version of the algorithm is the same as the forward version except for two small changes: During initialization, $\mathsf{dist}[t]$ is set to 0 instead of $\mathsf{dist}[s]$ and t is inserted into Q instead of s. In the main loop, when settling a vertex u, the backwards version scans all incoming edges (v, u) instead of all outgoing edges (u, v).

Label-Correcting Algorithm. The proof of the correctness and the complexity analysis given above both assume that the graph contains no negative edge weights. While this condition holds for some practical applications, such as calculating time-optimal routes, it is not fulfilled when using energy consumption as an edge weight, due to recuperation. While Dijkstra's algorithm still computes correct shortest paths when negative edge weights are allowed, provided that the graph contains no negative cycles, the worst-case complexity in this case is exponential in the size of the graph. This is caused by the fact that the label-setting property no longer holds: Without negative edge weights, when settling a vertex u, an edge to an already settled vertex v can never be relaxed, because $\mathsf{dist}[v] \leq \mathsf{dist}[u]$ holds. However, if the edge $e = (u, v)$ has a negative weight, the $\mathsf{dist}[u] + \omega(e)$ may become smaller than the current value of $\mathsf{dist}[v]$, which will cause v to be re-inserted into the queue. Thus, the algorithm becomes *label-correcting*. In addition, the stopping criterion used for the single-pair shortest path problem can no longer be used, because the distance of the target vertex t may decrease further even after it is settled.

Another algorithm for solving the single-source shortest path problem is the Bellman-Ford algorithm, which computes shortest paths with a polynomial worst-case complexity of $\mathcal{O}(nm)$ even if the graph contains negative edge weights. However, on typical road networks with only a small fraction of edges with a negative weight, Dijkstra's algorithm is usually faster than the Bellman-Ford algorithm because the number of vertices that are settled more than once remains low. In the next section, we discuss a technique called *potential shifting*, which uses an additional preprocessing step to remove the negative edge weights in the graph. This makes the worst-case complexity polynomial again and also re-enables the stopping criterion for the single-pair shortest path problem.

2.3.2 Potential Shifting

The potential shifting technique was first introduced by Donald B. Johnson in 1977 as part of an algorithm for solving the all-pairs shortest path problem [Joh77]. The idea behind the technique is to calculate a *potential function* $\Phi \colon V \to \mathbb{R}$ that assigns a *potential* to every vertex in V. The potentials are then used to define a new edge weight function ω'. The redefined edge weight function must fulfill two conditions: The edge weights assigned by ω' must be non-negative, i.e., $\omega'(e) \geq 0$ for all $e \in E$. Furthermore, the structure of the shortest paths may not change, i.e., a shortest path between two vertices $u, v \in V$ under ω must also be a shortest path under ω'.

The simplest way of obtaining a potential function is to choose an arbitrary *root* vertex r and calculate the shortest path distances from r to each vertex for the edge weight function ω.

This can be done with Dijkstra's algorithm (or the Bellman-Ford algorithm if polynomial worst-case complexity is desired). The potential function is then defined as follows for every vertex $v \in V$: $\Phi(v) \coloneqq \mathrm{dist}_\omega(r, v)$. Using this potential function, the new edge weight function is defined as follows for every edge $e = (u, v)$: $\omega'(e) = \omega(e) - \Phi(v) + \Phi(u)$. This edge weight function fulfills the two conditions outlined above:

Theorem 2.3. *Let $G = (V, E)$ be a graph with an edge weight function $\omega\colon E \to \mathbb{R}$ and a root vertex $r \in V$. For this root vertex we define a potential function $\Phi\colon V \to \mathbb{R}$ and an edge weight $\omega'\colon E \to \mathbb{R}$ as follows:*

$$\Phi(v) \coloneqq \mathrm{dist}_\omega(r, v)$$
$$\omega'((u, v)) \coloneqq \omega((u, v)) - \Phi(v) + \Phi(u).$$

The following two conditions are fulfilled for ω':

1. $\forall e \in E : \omega'(e) \geq 0$.
2. *Given two vertices $u, v \in V$ where v is reachable from u, every shortest u-v-path under ω is also a shortest u-v-path under ω'.*

Proof. The first claim can be proven by contradiction. Assume there is an edge $e = (u, v)$ such that $\omega'(e) < 0$. This means that $\Phi(v) > \Phi(u) + \omega(e)$. Substituting with the definition of Φ yields $\mathrm{dist}_\omega(r, v) > \mathrm{dist}_\omega(r, u) + \omega(e)$. This means that the shortest r-u-path extended by e is shorter than the shortest r-v-path, which contradicts the definition of a shortest path.

To prove the second claim, we observe that the weight of a path $P = (u = v_1, v_2 \ldots, v_k = v)$ under ω' does not depend on the potentials of its vertices except for u and v. The potential of a vertex v_i ($i \in [2, k-1]$) is added to $\omega'((v_i, v_{i+1}))$ and subtracted from $\omega'((v_{i-1}, v_i))$, so when the weights of all edges on the path are added together, all potentials except $\Phi(u)$ and $\Phi(v)$ cancel each other out. Thus, $\omega'(P) = \omega(P) - \Phi(v) + \Phi(u)$. Among all u-v-paths, $\Phi(u)$ and $\Phi(v)$ are constant, so the shortest u-v-path under ω remains the shortest u-v-path under ω'. $\square$

Because these two conditions are fulfilled, ω' is suitable as a replacement for ω. The first condition ensures that the label-setting property holds, which guarantees a polynomial running time for Dijkstra's algorithm and re-enables the stopping criterion for the single-pair shortest path problem. The second condition ensures that the shortest paths that are found under the new edge weight function are the same ones that would have been found with the old edge weight function. However, their length will differ from the length under the original edge weight function. We can calculate the original length as follows: If $\mathrm{dist}_{\omega'}(s, t)$ is the length of the shortest s-t-path under ω', then the length of that path under ω is $\mathrm{dist}_\omega(s, t) = \mathrm{dist}_{\omega'}(s, t) + \Phi(t) - \Phi(s)$.

In summary, potential shifting allows us the regain the advantages that were lost due to negative edge weights at the cost of a single preprocessing step that only needs to be performed once for a given graph and edge weight function.

2.4 Electric Vehicle Route Planning

Finding energy-optimal routes for electric vehicles on a road network is closely related to solving the shortest path problem with the edge weight $\mathrm{cons}\colon E \to \mathbb{R}$, which assigns to every edge the energy that is consumed on it. However, it is not identical because we have to take into account the constraints imposed by the vehicle's battery. The battery has a maximum capacity which we denote by M. While the vehicle may recuperate energy and recharge the battery on edges with a negative energy consumption, the battery's *state of charge* (SoC) may never exceed M. Once this limit is reached, any further energy that might otherwise be recuperated is lost. Likewise, the SoC may never become negative. Once the SoC reaches 0, the vehicle becomes stranded and cannot drive further without recharging the battery. In realistic applications it may be undesirable to allow the battery to be discharged entirely, to avoid getting stranded and to prevent damage to the battery. However, for the sake of simplicity we define the lower bound for the battery's SoC as 0, since this has no impact on the problem's complexity or the presented algorithms. The range of valid battery states is thus $B = [0, M]$.

We call a path *feasible* if the SoC remains in B for all vertices on the path. Naturally, we are only interested in feasible paths. To check if a path P is feasible, it is not sufficient to look at the total energy consumption $\operatorname{cons}(P)$ on the path, because this neglects two special cases: If the path contains an edge that causes the vehicle's SoC to drop below 0, the battery is *undercharged* and the path is not feasible. If the path contains an edge that would cause the vehicle's SoC to exceed M, the battery is *overcharged* and we need to limit its SoC to M. In order to handle these two special cases, we need to look at every vertex and edge in P separately.

The initial SoC at the source vertex s is given by $b(s)$. When traversing an edge $e = (s, v)$, we reduce $b(s)$ by $\operatorname{cons}(e)$, unless this would cause the battery to be overcharged, in which case we set the SoC to M. This yields for the SoC at v: $b(v) = \min(b(s) - \operatorname{cons}(e), M)$. If $b(v)$ is negative, this indicates that traversing e would cause undercharging and is not feasible for the initial SoC $b(s)$. By expanding this formula to an entire path we obtain the function $b(P, i)$, which computes the vehicle's SoC after traversing the first i vertices of the path $P = (v_1, v_2, \dots, v_k)$:

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

We thus call P *feasible* if $b(P, i) \in B$ holds for all $1 \leq i \leq k$.

Before we can give a definition of the electric vehicle route planning problem, we need to reconsider our definition of a shortest path. Before battery constraints were involved, we called a path $P = (s = v_1, v_2, \dots, v_k = t)$ the shortest s-t-path under an edge weight ω if it minimized $\omega(P)$. This is no longer possible because $\operatorname{cons}(P)$ no longer represents the total amount of consumed energy on P (although it is still a lower bound for the consumed energy). Instead, the total amount of consumed energy is given by $b(s) - b(t)$, which is equivalent to $b(P, 1) - b(P, k)$. Since $b(s)$ is given as an input to our problem and is constant, this is equivalent to maximizing $b(t)$. We thus call P the shortest s-t-path if it maximizes $b(t)$. With this we can give a definition of our problem:

Definition 2.4. Energy-Optimal Electric Vehicle Route Planning (EEVR)
We are given a graph $G = (V, E)$, an energy consumption function $\operatorname{cons}\colon E \to \mathbb{R}$*, source and target vertices $s, t \in V$, a range of valid battery states $B = [0, M]$ and an initial state of charge $b(s)$. We ask for the shortest feasible path $P = (s = v_1, v_2, \dots, v_k = t)$ in G, i.e., the path that maximizes $b(t)$ among all s-t-paths for which $b(P, i) \in B$ for all $1 \leq i \leq k$.*

There are two possible approaches to adapting Dijkstra's algorithm to solve this problem. One is to use the current SoC as the key for the priority queue instead of the distance (which is equivalent to the consumed energy in this scenario). Instead of extracting the element with the lowest key in each iteration of the loop, the element with the highest key is extracted (since this is the vertex with the lowest consumed energy). When relaxing an edge $e = (u, v)$, the battery constraints are checked explicitly: If $b(u) - \mathrm{cons}(e) < 0$, traversing the edge is not feasible and it is simply discarded. If $b(u) - \mathrm{cons}(e) > M$, $b(v)$ is simply set to M to prevent overcharging.

The second approach was introduced by [EFS11] and models the battery constraints implicitly as part of the edge weight function. This can be achieved by replacing the scalar edge weights with *energy consumption functions* that map each valid SoC to the energy that is consumed when traversing the edge with that SoC. While this approach seems more complicated at first, it makes it easier to solve *profile queries*, which are queries that do not supply the initial SoC $b(s)$ at the source vertex as an input and instead ask for the shortest feasible path for every possible value of $b(s)$. These queries will become important when incorporating charging stations into our problem and are discussed in Section 2.4.3.

2.4.1 Energy Consumption Functions

An *energy consumption function* is a function $c\colon B \to \mathbb{R} \cup \{\infty\}$ that maps SoC to energy consumption. For an edge $e = (u, v)$, the corresponding energy consumption function c_e describes for each possible SoC b the amount of energy that is consumed on e if the vehicle arrives at u with a SoC of b. The special value ∞ indicates that the SoC is not sufficient for traversing the edge. This energy consumption function already takes battery constraints into account, so an algorithm using these functions as edge weights will not need to handle over- and undercharging explicitly.

Defining the energy consumption function c_e for an edge $e = (u, v)$ is fairly simple: If no over- or undercharging occurs, the function simply returns the original edge weight $\mathrm{cons}(e)$. Undercharging occurs if the SoC $b(u)$ at u is smaller than the energy consumption of the edge $\mathrm{cons}(e)$, i.e., if $b(u) - \mathrm{cons}(e) < 0$. In this case, the SoC is insufficient for traversing the edge and the function returns ∞ to indicate this. Overcharging occurs if the SoC would exceed M after traversing the edge, i.e., if $b(u) - \mathrm{cons}(e) > M$. Since the vehicle can only recuperate energy until the battery is full, the energy consumption in this case is $b(u) - M$. In summary, this leads to the following definition for c_e:

$$c_e(b) = \begin{cases} \infty & \text{if } b - \mathrm{cons}(e) < 0 \\ \mathrm{cons}(e) & \text{if } b - \mathrm{cons}(e) \in B \\ b - M & \text{if } b - \mathrm{cons}(e) > M. \end{cases}$$

An example of an energy consumption function can be found in Figure 2.1. Using these energy consumption functions, we can now calculate the SoC at v after traversing e as $b(v) = b(u) - c_e(b(u))$. The condition $b(v) \in B$ will hold unless $b(u)$ is not sufficient to traverse e, in which case $b(v)$ will have the special value $-\infty$. Furthermore, as observed in [EFS11], this definition fulfills the *FIFO property*: For two SoC values $b_1, b_2 \in B$ with $b_1 \leq b_2$, $b_1 - c_e(b_1) \leq b_2 - c_e(b_2)$ is always fulfilled. This means that if the vehicle starts with a higher SoC at u, it cannot end up with a lower SoC at v.

We observe that c_e is a piecewise linear function. Each of the three parts of the function definition corresponds to one interval. The value of c_e in the first two intervals is ∞ and $\mathrm{cons}(e)$, respectively, so in these intervals c_e is constant and has a slope of 0. In the third interval, c_e is a linear function with slope 1. The size and position of the three intervals depends on the value of $\mathrm{cons}(e)$, and some of the intervals may disappear entirely:

- If $\mathrm{cons}(e)$ is negative, the first part of the function definition disappears. The function c_e then consists of the two intervals $[0, M+\mathrm{cons}(e))$ and $[M+\mathrm{cons}(e), M]$ and is represented by the two supporting points $(0, \mathrm{cons}(e), 0)$ and $(M+\mathrm{cons}(e), \mathrm{cons}(e), 1)$.

- If $\mathrm{cons}(e)$ is 0, only the middle part of the function definition remains. The function c_e then consists only of one interval $[0, M]$ and is represented by the supporting point $(0, 0, 0)$.
- If $\mathrm{cons}(e)$ is positive, the last part of the function definition disappears. The function c_e then consists of the two intervals $[0, \mathrm{cons}(e))$ and $[\mathrm{cons}(e), M]$ and is represented by the two supporting points $(0, \infty, 0)$ and $(\mathrm{cons}(e), \mathrm{cons}(e), 0)$.

Every energy consumption function can be represented with only three static values cost, minIn and maxOut: cost is the energy consumption if neither over- nor undercharging occurs. minIn is the minimal SoC that is needed for traversing the edge. maxOut is the maximal SoC that can be achieved after traversing the edge. For an edge e, these values can all be calculated from $\mathrm{cons}(e)$:

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

With cost, minIn and maxOut, the energy consumption c_e for an edge e can be redefined as

$$c_e(b) = \begin{cases} \infty & \text{if } b < \mathrm{minIn}_e\\ \mathrm{cost}_e & \text{if } b - \mathrm{cost}_e < \mathrm{maxOut}_e\\ b - \mathrm{maxOut} & \text{otherwise.}\end{cases}$$

If we replace the static edge weights with these energy consumption functions, we can use Dijkstra's algorithm directly to solve the EEVR problem. When relaxing an edge $e = (u, v)$, we evaluate the energy consumption function c_e at the current SoC $b(u)$ to obtain the energy consumption for e without having to handle over- and undercharging explicitly. In the version presented in Algorithm 2.1, Dijkstra's algorithm stores only the tentative distance from s for each vertex v in $\mathsf{dist}[v]$, but not the vehicle's SoC at v, which we need to evaluate the energy consumption function. To remedy this, we can either extend the algorithm by another array $\mathrm{SoC}[\cdot]$ that stores the tentative SoC at each vertex or calculate the SoC implicitly as $b(v) = b(s) - \mathsf{dist}[v]$.

Linking. Just like with scalar edge weight functions, we can extend our definition of energy consumption functions to apply to entire paths. However, this is no longer as simple as calculating the sum of all edge weights since the total energy that is consumed on a path may be affected by over- and undercharging. Instead we define the *link* operation $\circ$. Given two edges $e, f \in E$, the link operation takes their energy consumption functions c_e and c_f and calculates the linked energy consumption function $c_{e\circ f}$, which describes the energy consumption for traversing e and f successively. Formally it is defined as $c_{e\circ f}(b) \coloneqq c_f(b - c_e(b))$. Like the energy consumption function for a single edge, it can be represented with the three values $\mathrm{cost}_{e\circ f}$, $\mathrm{minIn}_{e\circ f}$, $\mathrm{maxOut}_{e\circ f}$. They can be calculated from the values for c_e and c_f as follows:

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

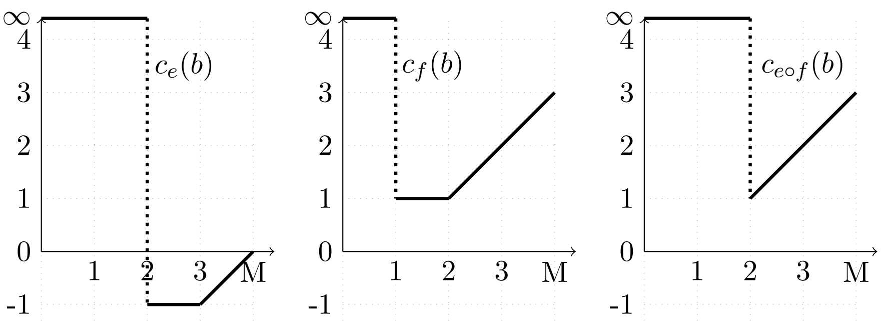

Figure 2.1: An example of the link operation for energy consumption functions. The maximal possible SoC is defined as $M := 4$ in this example. The function c_e is defined by $\text{cost}_e = -1$, $\text{minIn}_e = 2$ and $\text{maxOut}_e = 4$. The function c_f is defined by $\text{cost}_f = 1$, $\text{minIn}_f = 1$ and $\text{maxOut}_f = 1$. Linking the two functions results in $c_{e \circ f}$, which is defined by $\text{cost}_{e \circ f} = 1$, $\text{minIn}_{e \circ f} = 2$ and $\text{maxOut}_{e \circ f} = 1$.

Obviously, e and f can only be traversed if the initial SoC is sufficient for traversing e. Thus, minIn_e is a lower bound for $\text{minIn}_{e \circ f}$. Additionally, after traversing e, there must still be enough SoC left to also traverse f, which means $\text{minIn}_f + \text{cost}_e$ is also a lower bound. In total, $\text{minIn}_{e \circ f}$ is the maximum of these two lower bounds. The definition of $\text{maxOut}_{e \circ f}$ is similar: After traversing both e and f, the SoC cannot be higher than the maximum SoC that is possible after traversing f alone. However, it also cannot be higher than the SoC that is achieved after traversing f with the maximum SoC that is achievable after traversing e, so in total $\text{maxOut}_{e \circ f}$ is the minimum of maxOut_f and $\text{maxOut}_e - \text{cost}_f$.

Usually, $\text{cost}_{e \circ f}$ will simply be the sum of cost_e and cost_f. However, there is one exception: If traversing e and f successively will always cause either over- or undercharging to occur, regardless of the initial SoC, the lowest possible energy consumption is higher than $\text{cost}_e + \text{cost}_f$. An example of this is shown in Figure 2.1. Here the middle part of the function definition disappears, causing either over- or undercharging to occur for every SoC value. The minimal energy consumption is therefore not $\text{cost}_e + \text{cost}_f = 0$. Rather, the minimal amount of energy is consumed at the point where the first and third part of the function definition meet, i.e., at the intersection of the two lines $b = \text{minIn}$ and $\text{cost} = b - \text{maxOut}$. Thus, $\text{minIn}_e - \text{maxOut}_f$ is a lower bound for $\text{cost}_{e \circ f}$.

The link operation can be easily applied to paths by simply iterating over the edges of a path P and linking their energy consumption functions together in the order in which they occur in P. In fact, since the link operation is associative, the energy consumption function for P can be calculated by splitting P into any arbitrary configuration of subpaths, calculating the energy consumption functions for these subpaths and then linking them together in the correct order.

Since linked energy consumption functions are still covered by the same function definition as single-edge energy consumption functions, just with different values for cost, minIn and maxOut, they are also piecewise linear functions with at most three intervals. For single edges, at least one case of the function definition always disappeared. This is no longer necessarily the case for linked energy consumption functions, so we may need three supporting points to represent them.

2.4.2 Potential Shifting

In order to guarantee a polynomial running time and enable the stopping criterion for the EEVR problem, we need to adapt the potential shifting technique to work with our new energy consumption functions, which was first done in [EFS11].

We cannot simply use the shifted edge weight functions from Section 2.3.2 because instead of static edge weights $\text{cons}(e)$ we now have energy consumption functions c_e. Accordingly, for every edge $e = (u, v)$ we need to replace the energy consumption function c_e with a shifted function c'_e. Like c_e, this function needs to map the SoC at u to the (shifted) energy that is consumed when traversing e. For a suitable potential function $\Phi\colon V \to \mathbb{R}$, we define our shifted energy consumption function as follows: $c'_e(b) := c_e(b) - \Phi(v) + \Phi(u)$. This definition is equivalent to the one for static edge weights in Section 2.3.2, except that the static edge weight $\omega(e)$ has been replaced with the energy consumption function c_e.

In order to be able to evaluate our shifted energy consumption function, we need to know the SoC at u, which Dijkstra's algorithm does not save by default. We thus add an array $\text{SoC}[\cdot]$ that holds for each vertex v the vehicle's SoC at v, similar to the array $\mathsf{dist}[\cdot]$, which holds the energy consumed so far. Whenever an edge $e = (u, v)$ is relaxed, we update the SoC at v: $\text{SoC}[v] = \text{SoC}[u] - c_e(\text{SoC}[u])$. Note that we use the original energy consumption function c_e to update $\text{SoC}[v]$, whereas the shifted energy consumption function c'_e is used to update $\mathsf{dist}[v]$. The reason for this is that c' does not reflect the actual energy that is consumed by the vehicle, so if we were to use it to update $\text{SoC}[\cdot]$, it would no longer reflect the actual SoC of the vehicle, which we need to know to evaluate the energy consumption function.

Now we need to choose a potential function. As it turns out, the one we defined in Section 2.3.2, which was obtained by choosing an arbitrary root vertex r and calculating the distances to every other vertex using the original static edge weights, is still suitable. We prove this and the suitability of our shifted energy consumption function in the following theorem:

Theorem 2.5. *Let $G = (V, E)$ be a graph with an edge weight function* $\text{cons}\colon E \to \mathbb{R}$ *and a root vertex $r \in V$. For this root vertex we define a potential function $\Phi\colon V \to \mathbb{R}$ as follows:*

$$\Phi(v) := \text{dist}_{\text{cons}}(r, v).$$

For every edge $e \in E$, let $c_e\colon B \to \mathbb{R} \cup \{\infty\}$ be the energy consumption function corresponding to the edge weight $\text{cons}(e)$*. We define the shifted energy consumption function $c'_e\colon B \to \mathbb{R} \cup \{\infty\}$ as follows:*

$$c'_e(b) := c_e(b) - \Phi(v) + \Phi(u).$$

The following two conditions are fulfilled for c'_e:

1. *$\forall e \in E, b \in B : c'_e(b) \geq 0$.*
2. *Given two vertices $u, v \in V$ where v is reachable from u, every shortest u-v-path under c is also a shortest u-v-path under c'.*

Proof. In order to prove the first claim, we observe that $\text{cons}(e) \leq c_e(b)$ holds for every edge $e \in E$ and every SoC $b \in B$. In other words, the energy that is consumed when traversing an edge is never lower than its weight. There are only three cases to consider: If undercharging occurs, $c_e(b)$ returns ∞. Overcharging occurs if $b - M > \text{cons}(e)$ holds, and in this case, $b - M$ is returned. If neither over- nor undercharging occur, $\text{cons}(e)$ itself is returned.

With this, we can prove the first claim by contradiction. Assume there is an edge $e = (u, v)$ and a SoC $b \in B$ such that $c'_e(b) < 0$. This means that $\Phi(v) > \Phi(u) + c_e(b)$. Using the above observation, this becomes $\Phi(v) > \Phi(u) + \text{cons}(e)$. This was already proven to be a contradiction in Theorem 2.3.

The proof for the second claim is also analogous to Theorem 2.3: The energy consumption of a path $P = (u = v_1, v_2 \dots, v_k = v)$ under the shifted energy consumption function c_P of P only depends on the potentials of u and v and is given by $c'_P(b) = c_P(b) - \Phi(v) + \Phi(u)$, where $b \in B$ is the SoC at u. Since $\Phi(u)$ and $\Phi(v)$ are constant among all u-v-paths, the shortest u-v-path under c' remains the shortest u-v-path under c. □

As the theorem proves, the shifted energy consumption function c'_e as defined above removes all negative edge weights and does not change the structure of the shortest paths, so replacing the original energy consumption function c_e with it will re-enable our stopping criterion and guarantee a polynomial running time.

2.4.3 Profile Queries

In addition to regular queries where the initial SoC $b(s)$ is given as an input, we also want to be able to solve *profile queries*, where we ask for the shortest feasible path for every possible value of $b(s)$. Since the range of valid battery states $B = [0, M]$ is a continuous set and thus has infinitely many members, we cannot simply run a separate query for every possible value of $b(s)$. Instead, our output must be a function that maps each possible initial SoC to the energy that is consumed on the shortest feasible path for that SoC. Mapping initial SoC to consumed energy is exactly what the energy consumption functions introduced in the previous section do, so by using them as vertex labels, we can easily adapt our algorithm to solve profile queries.

Instead of maintaining a single shortest distance from s for each vertex v, we now store an energy consumption function that maps each initial SoC $b(s)$ to the shortest s-v-distance for that value of $b(s)$. We call this function the *energy consumption profile* of v. Accordingly, we replace the array $\mathsf{dist}[\cdot]$, which holds the distances from s, with an array of energy consumption profiles $\text{profile}[\cdot]$. In the initialization phase of the algorithm, we initialize the profile of each vertex v with a constant function $\text{profile}[v](b) = \infty$ that maps every value of $b(s)$ to ∞, to signify that no path to the vertex has been found yet. The profile of s is set to a constant function $\text{profile}[s](b) = 0$.

The algorithm now maintains a priority queue Q of vertex profiles. As key, we use the first non-infinite energy consumption value of the profile. When relaxing an edge $e = (u, v)$, we link the profile of u with the energy consumption function of e to obtain a new candidate for the profile for v: $\text{newProfile} \leftarrow \text{profile}[u] \circ c_e$. Now we need to compare this candidate to the current profile at v to see which one represents the shorter path. This is no longer as simple as it was for a regular query, where we only needed to compare two energy consumption values and pick the lower one of the two. Now, however, we need to compare two functions. It is possible that the new profile is superior than the current profile for some SoC values yet inferior for others. An example of this can be seen in Figure 2.2.

In order to update the profile at v, we need to *merge* it with the newly calculated profile. For each initial SoC $b(s)$, we must evaluate both profiles, compare the energy consumptions and pick the lower one of the two. Mathematically, this means computing the lower envelope of the two functions. Thus, we define a *merge* operation that does exactly this for two given profiles p and q: $\text{merge}(p, q) := \min(p, q)$. We can calculate the result of this operation

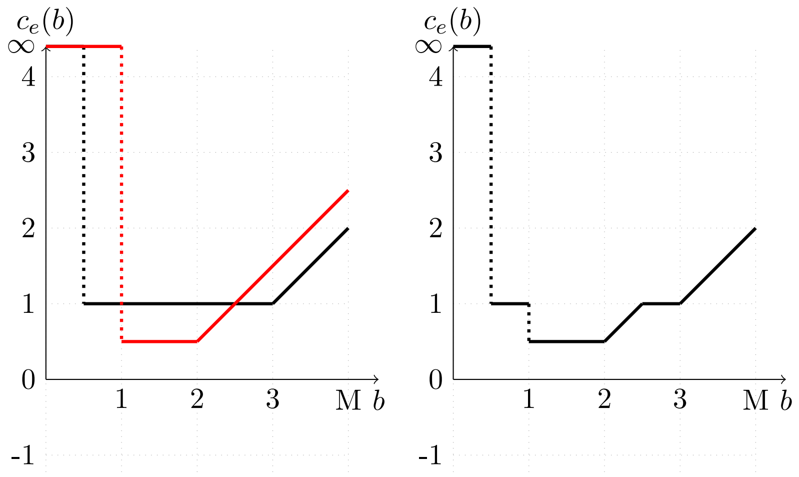

Figure 2.2: An example of the merge operation for profile functions. To the left are two profile functions that represent different paths to the same vertex. The black function is superior for some SoC values, while the red function is superior for others. To the right is the profile that results from merging the red and black profiles, which is the lower bound of the two functions. Note that while the original profiles only had three supporting points each, the merged profile has six. Of these, the supporting point at $(2.5, 1)$ appears in neither of the original two profiles. It was added because it lies at the intersection of a function segment with slope 0 in the black profile and a function segment with slope 1 in the red profile.

by performing a coordinated linear sweep over the supporting points of both functions. An example of the merge operation is shown in Figure 2.2. In our algorithm, we can now use this operation every time an edge $e = (u, v)$ is relaxed. After computing newProfile $\leftarrow$ profile$[u] \circ c_e$, we merge it with the profile of v: profile$[v] \leftarrow$ merge(profile$[v]$, newProfile). If profile$[v]$ changed as a result of this merging operation, we recalculate its key and update the priority queue.

So far, energy consumption functions were piecewise linear functions that could be represented with at most three supporting points as well as with the three values cost, minIn and maxOut. However, as seen in Figure 2.2, the merge operation may increase the number of supporting that are needed to represent the function. For this reason, it is no longer possible to represent merged profiles with the values cost, minIn and maxOut. Instead, we have to represent the profiles by their supporting points and perform the link and merge operations by iterating over the supporting points of the functions.

Complexity. The energy consumption profiles can now represent multiple paths for different initial SoCs, but the key for the priority queue is still a single value. This means it is possible for the profile of a vertex v to be modified by a merge operation without its key being changed. This in turn means that the label-setting property no longer holds, even after potential shifting. The running time of the algorithm now depends on the complexity of the profile functions [BDPW13].

Path Retrieval. While a profile query gives us shortest path distances for every possible initial SoC, we may still want to retrieve the shortest path for a specific initial SoC $b(s)$. In order to do this, we must extend the vertex profiles by parent pointers. However, a single parent pointer per vertex v no longer suffices, since the shortest s-v-path may be different depending on the initial SoC $b(s)$. Instead we need a parent pointer for every point in the profile of v where the path represented by the profile changes. This can only occur at supporting points, so we extend the profiles to hold a parent pointer for each supporting point.

When relaxing an edge $e = (u, v)$, we link the profile of u with the energy consumption function of e. In the resulting profile, we can set the parent pointer to u for all supporting points, since this was the last visited vertex regardless of the initial SoC. When merging the new profile with the current profile of v, there are two types of supporting points in the merged profile: those that appeared in the original profiles and those that were added at intersection points. For the former, we can simply retain the parent pointers of the original supporting points. At a supporting point, a function segment with slope 0 from one profile p intersects with a function segment with slope 1 from the other profile b. To the left of the intersection point, the value of b is lower than the value of a, but at the intersection point it overtakes a and the value of a becomes lower. Since the new supporting point represents the interval to the right, where a is lower, we must use the parent pointer from a. This is the parent pointer belonging to the last supporting point of a to the left of the intersection point.

When asked for the shortest path from s to a target vertex t for an initial SoC b, we must first identify the correct parent pointer in the profile of t. This is the parent pointer of the supporting point whose corresponding interval includes b. Then we follow this parent pointer to find the parent vertex u of t. At u, we must again find the correct supporting point. All vertex profiles in the graph map the initial SoC from the same vertex s to the energy that must be consumed in order to reach the vertex that the profile belongs to. Therefore, we again look up the supporting point in the profile of u whose interval includes b and follow its parent pointer. We continue doing this until we reach s.

Backwards Search. In order to solve single-destination profile queries, we can again reverse the search direction of our algorithm by scanning incoming edges instead of outgoing edges in the main loop. When relaxing such an edge $e = (v, u)$, we link the energy consumption function of e with the profile of u, but in reverse order compared to a forward search: $\text{newProfile} = c_e \circ \text{profile}[u]$. We set the parent pointers of newProfile to u for all supporting points.

While the algorithm itself remains unchanged except for reversing its search direction, retrieving the shortest s-t-path for a given source vertex s and an initial SoC b is now slightly more complicated: In the forward version of the algorithm, the profile of a vertex v represents the path from the source vertex s to v, where the source vertex s is the same for all profiles. This means that when retrieving a path, we can look up all profiles at the same initial SoC b and find the corresponding supporting point. In the backwards version, however, this is no longer the case: The profiles now all share the same target vertex t, but the source vertex is different for every profile. Therefore, we need to update the initial SoC during backtracking.

We start the path retrieval at s with an initial SoC $b(s)$. We evaluate the profile of s at the point $b(s)$ to find the corresponding supporting point and its parent pointer. We then follow the parent pointer to the next vertex u. After traversing the edge $e = (s, u)$, our new initial SoC is $b(u) = b(s) - c_e(b(s))$. We use this new initial SoC to find the corresponding supporting point and the next parent pointer. We continue doing this until we reach t, always updating the initial SoC b by evaluating the energy consumption function of the traversed edge at b and subtracting the resulting energy consumption from b.

3. Problem Statement

In the previous chapter we introduced an algorithm that finds energy-optimal routes for electric vehicles by adapting the shortest path problem. This was achieved by replacing the static edge weights with energy consumption functions. However, the length of the routes that can be found with this approach is limited by the capacity of the battery. In order to calculate long-distance routes, we need to consider *charging stations* where the battery can be recharged. In this thesis we consider three types of charging stations:

- Regular charging stations allow the battery to be charged to any desired SoC within the battery's capacity $B = [0, M]$. It may be recharged to full capacity or only partially.
- Superchargers are a special type of charging station that recharge the battery much faster than regular charging stations, but due to technical limitations, the battery can only be charged up to 80% of its capacity. This means that the range of SoCs to which the battery can be charged is $[0, 0.8 \cdot M]$.
- Battery swapping stations do not recharge the battery but rather exchange it with another, fully charged battery. This means that the SoC to which the battery is charged cannot be chosen; it is either "charged" to M or not at all.

For a graph $G = (V, E)$ that represents a road network, we model the charging stations as subset $\mathrm{CS} \subseteq V$ of the set of vertices. Additionally, we define a function $\mathrm{type}\colon \mathrm{CS} \to \{\mathrm{REG}, \mathrm{SC}, \mathrm{BSS}\}$ that assigns a type to each charging station: REG stands for regular charging stations, SC for superchargers and BSS for battery swapping stations.

In order to completely describe a route that uses charging stations, we need to know not only where to charge but also how much. For an energy-optimal path P, we define a *charging distribution function* $\mathrm{cd}\colon \mathrm{CS} \cap P \to B$ that assigns to every charging station encountered on the path the amount of energy that is charged there. Depending on the course of the path, this distribution might be ambiguous: We might encounter more charging stations than we actually need to use in order to reach the target t. Since charging stops are time-consuming, especially for electric vehicles, it is usually desirable to use as few charging stations as possible, so we introduce an additional requirement that the charging distribution minimizes the number of used charging stations on P. There may still be multiple solutions that fulfill this requirement, for example if two charging stations are close together and charging at either of them is sufficient to reach the target, but in this case we are satisfied with either solution.

With the addition of charging stations, our definition of a *feasible* path changes. As before, the range of valid battery states is denoted by $B = [0, M]$ and the SoC at the source vertex s is denoted by $b(s)$. We consider a path $P = (s = v_1, b_2, \dots, v_k = t)$ feasible if the SoC $b(v)$ at every vertex $v \in V$ in the path remains in B. Previously, the SoC at a vertex v_i $(1 < i \leq k)$ could be calculated from the SoC of the previous vertex v_{i-1} by subtracting the energy consumed on the edge (v_{i-1}, v_i). This is still the case, but if v_i contains a charging station, we also have to consider the effects of recharging. For a given charging distribution $\mathrm{cd}\colon \mathrm{CS} \cap P \to B$ we thus define the function $b(P, i)$, which computes the SoC after traversing the first i vertices of P:

$$b(P,i) \coloneqq \begin{cases} b(v_1) & \text{if } i = 1 \\ \min(b(P, i-1) - \mathrm{cons}(v_{i-1}, v_i), M) & \text{if } v_i \notin \mathrm{CS} \\ \min(b(P, i-1) - \mathrm{cons}(v_{i-1}, v_i), M) + \mathrm{cd}(v_i) & \text{otherwise.} \end{cases}$$

The first two cases are identical to the previous definition. In the third case, the amount of energy charged at v_i is added to $b(P, i)$. As before, we consider P feasible if $b(P, i) \in B$ holds for all $1 \leq i \leq k$.

The definition of a *shortest* path must also be adapted to take charging stations into account: Minimizing $\mathrm{cons}(P)$ is again not sufficient because this would neglect the effects of over- and undercharging. However, we can also no longer maximize $b(t)$ like before, since this would allow us to calculate routes with arbitarily large detours as long as the amount of energy consumed between the last charging station to t is minimized. Indeed, the total amount of consumed energy on P is no longer simply given by $b(s) - b(t)$, since this ignores the energy that was recharged on P. Instead, it can now be calculated as

$$c_P(b(s)) = b(s) - b(t) + \sum_{\mathrm{cs} \in \mathrm{CS} \cap P} \mathrm{cd}(\mathrm{cs}).$$

The total amount of consumed energy is the amount of energy that was charged on P plus the difference between the initial SoC at s and the eventual SoC at t. This is the value that a shortest path must minimize. With these considerations, we can now define the Energy-Optimal Electric Vehicle Routing with Recharging problem:

Definition 3.1. Energy-Optimal Electric Vehicle Routing with Recharging (EEVRC)
We are given a graph $G = (V, E)$, an energy consumption function $\mathrm{cons}\colon E \to \mathbb{R}$, source and target vertices $s, t \in V$, a range of valid battery states $B = [0, M]$, an initial state of charge $b(s)$, a set of charging stations $\mathrm{CS} \subseteq V$ and a function $\mathrm{type}\colon \mathrm{CS} \to \{\mathrm{REG}, \mathrm{SC}, \mathrm{BSS}\}$ that assigns a type to each charging station.

We ask for a path $P = (s = v_1, v_2, \dots, v_k = t) \in G$ and a charging distribution function $\mathrm{cd}\colon \mathrm{CS} \cap P \to B$ so that P is the shortest feasible s-t-path, i.e., the path that minimizes $c_P(b(s))$ among all s-t-paths for which $b(P, i) \in B$ for all $1 \leq i \leq k$. We also require that number of used charging stations $|\{\mathrm{cs} \in \mathrm{CS} \cap P \mid \mathrm{cd}(\mathrm{cs}) > 0\}|$ is minimized.

So far no algorithm exists for solving EEVRC, but there are approaches that solve similar problems: Zündorf [Zün14] introduced an algorithm that finds time-optimal routes for electric vehicles while considering all three types of charging stations introduced above. This algorithm makes use of charging functions that map charging time to charged energy and incorporates them into a multi-objective variation of Dijkstra's algorithm in order to solve the problem. The algorithm introduced in Chapter 4 adapts concepts from this algorithm to find energy-optimal routes.

An algorithm developed by Storandt and Funke [SF12] also computes energy-optimal routes for electric vehicles, but only considers battery swapping stations as a method of recharging. It makes use of an precalculated auxiliary graph Q in which the shortest path between every pair of battery swapping stations $(c, d) \in \mathrm{CS}$ is contracted into a single edge with the link operation. It then answers an s-t-query in three steps: First it calculates the set $\mathrm{CS}_s \subseteq \mathrm{CS}$ of all battery swapping stations that can be reached directly from s with a Dijkstra query. Next, it calculates the set of all battery swapping stations $\mathrm{CS}_t \subseteq \mathrm{CS}$ from which t can be reached directly with a single-destination backwards query. As a last step, it completes the shortest path query by finding the remaining connection between CS_s and CS_t in Q with another Dijkstra query. This approach is adapted for other types of charging stations in Chapter 6.

4. Charging Label Algorithm

In this chapter, we will develop a first algorithm for solving the Energy-Optimal Electric Vehicle Routing with Recharging problem. As a basis, we will use the extension of Dijkstra's algorithm that was introduced in Chapter 2.4 for solving the EEVR problem. This algorithm already handles the battery constraints of the electric vehicle, but it does not yet take charging stations into account. In order to handle charging stations, we must decide how much energy to charge every time we encounter one.

At first, it might seem that an optimal approach is to always charge as much energy as possible when a vertex v with a charging station is encountered. This maximizes the SoC at v and therefore the distance that the vehicle can travel before having to recharge again. However, this neglects the possible effects of overcharging in the further course of the path: Consider an outgoing edge $e = (v, u)$ from v with an energy consumption $\text{cons}(e) = -1$. If the battery is charged to the maximum SoC M at v, the vehicle cannot recuperate any energy and the effective consumption is $c_e(M) = 0$. If the battery is charged to $M - 1$, however, the energy can be recuperated and the effective consumption is $c_e(M - 1) = -1 < c_e(M)$. Since we want to minimize the total energy consumption, it becomes clear that we cannot charge to M at v. We also do not know if $M - 1$ is the optimal SoC to charge to, since there might be another edge after e on which overcharging occurs for a SoC of $M - 1$ at v as well. The optimal amount of energy to charge depends on the further course of the path, which is not yet known by the time the search reaches v, so we must postpone the decision.

If we want to postpone the decision of how much energy to charge at v, that means we need to consider every possible amount of energy that may be charged. Since the amount of energy to charge is chosen from the continuous set $B = [0, M]$ of valid battery states, there are infinitely many possibilities that all need to be handled at once. A similar issue was encountered for the time-optimal version of the problem discussed in [Zün14], where it was resolved by introducing piecewise linear functions called *SoC functions* which mapped charging time to achieved SoC. We try to adapt this approach for the energy-optimal version of the problem.

4.1 Distance and SoC Functions

So far our algorithm uses the two arrays $\mathsf{dist}[\cdot]$ and $\text{SoC}[\cdot]$ to store the tentative energy consumption and SoC for each vertex, respectively. These single values now have to be

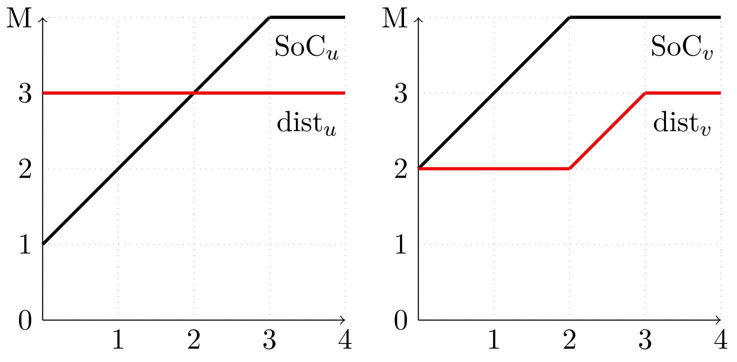

Figure 4.1: An example of SoC and distance functions. To the left are the SoC and distance function at a vertex u that contains a charging station. The vehicle reaches u with a SoC of 1 and a total energy consumption of 3. The SoC function SoC_u increases linearly as more energy is charged, until the maximum battery capacity $M := 4$ is reached, at which point it becomes constant. The energy consumption is unaffected by this, so dist_u is constant. To the right are the SoC and distance function after traversing an edge (u, v) with a consumption of -1. In the interval $[0, 2]$, this causes the SoC function SoC_v to be shifted upwards by one unit and the distance function dist_v to be shifted downwards by one unit. For higher amounts of charged energy, overcharging occurs because the SoC at u is too high to recuperate all energy. Therefore, SoC_v becomes constant while dist_v increases linearly.

replaced with functions. Thus, every vertex v now has a *distance function* $\mathrm{dist}_v : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$, which maps the amount of energy charged so far to the amount of energy consumed to reach v, and a *SoC function* $\mathrm{SoC}_v : \mathbb{R}_{\geq 0} \to B$, which maps charged energy to the current SoC at v. Examples of these functions are shown in Figure 4.1.

Before encountering a charging station, the only valid amount of charged energy is 0, so the functions dist_v and SoC_v are only defined at one single point, where they have the same values $\mathsf{dist}[v]$ and $\mathrm{SoC}[v]$ as in the original algorithm without functions. We can model dist_v and SoC_v as piecewise linear functions with the single supporting point $(0, \mathsf{dist}[v], 0)$ and $(0, \mathrm{SoC}[v], 0)$, respectively. In our definition of piecewise linear functions, the last interval of the function (which in this case is the only interval) is open-ended, i.e., it extends to ∞. However, in this case we only want to consider the first point of the interval as a valid point, since it represents the only valid amount of charged energy so far. Therefore, we establish the following convention for distance and SoC functions: In the last interval of the function, only the supporting point is considered as a valid point; the rest of the interval is ignored.

Once we encounter the first charging station, the range of possible values for charged energy expands from the single point to an entire interval. The size of the interval depends on the current SoC $\mathrm{SoC}_v(0)$ before charging. If $b_{\max} \in B$ is the maximum SoC that the battery can be recharged to at v (M for regular charging stations and $0.8 \cdot M$ for superchargers), the interval of possible values for charged energy is $[0, b_{\max} - \mathrm{SoC}_v(0)]$. Inside this interval, the achieved SoC increases linearly, reaching M at the endpoint. The amount of consumed energy is not affected by charging, so dist_v remains constant. Our functions are now represented by the supporting points $(0, \mathrm{dist}_v(0), 0)$ and $(b_{\max} - \mathrm{SoC}_v(0), \mathrm{dist}_v(0), 0)$ for dist_v, as well as $(0, \mathrm{SoC}_v(0), 1)$ and $(b_{\max} - \mathrm{SoC}_v(0), b_{\max}, 0)$ for SoC_v. Again, we ignore the second interval except for its supporting point, which represents the last valid amount of charged energy.

When traversing an edge $e = (u, v)$, we can use the energy consumption function c_e to calculate dist_v and SoC_v. This scenario is shown in Figure 4.1. For each amount of charged energy x, we evaluate c_e with the corresponding SoC $\mathrm{SoC}_u(x)$ at u to obtain the energy consumption on e:

$$\begin{aligned}\mathrm{dist}_v(x) &= \mathrm{dist}_u(x) + c_e(\mathrm{SoC}_u(x))\\ \mathrm{SoC}_v(x) &= \mathrm{SoC}_u(x) - c_e(\mathrm{SoC}_u(x)).\end{aligned}$$

If we encounter another charging station $v \in \mathrm{CS}$ on the path, we need to know the amount of energy x that was charged before reaching v so that we can extend the distance and SoC functions by the interval $[x, x + b_{\max} - \mathrm{SoC}_v(x)]$. Since we do not know the value of x yet and cannot determine it before the entire path has been found, we must consider every supporting point as a possible value for x. In the case of superchargers, we can ignore supporting points where the achieved SoC is already higher than $0.8 \cdot M$, since the battery cannot be recharged at v in this case.

For every supporting point x where $\mathrm{SoC}_v(x) \leq b_{\max}$ we define new functions dist_v^x and SoC_v^x that represent the energy consumption and SoC for the case where we charge to x before v and then switch over to the new charging station at v. These functions are constructed similarly to the functions for the first charging station: For dist_v^x, we extend dist_v by the supporting point $(x + b_{\max} - \mathrm{SoC}_v(x), \mathrm{dist}_v(x), 0)$. For SoC_v^x, we extend SoC_v by the supporting point $(x + b_{\max} - \mathrm{SoC}_v(x), b_{\max}, 0)$ and also change the slope of the previous supporting point $(x, \mathrm{SoC}_v(x), 0)$ to 1 to signify that the battery can be recharged.

If v is a battery swapping station, we only have the option of either swapping the battery or not at each supporting point x. If we choose to swap the battery, the amount of charged energy is now fixed at $x + M - \mathrm{SoC}_v(x)$ and we can discard all other possibilities. Thus, for each supporting point x, we define dist_v^x as the function represented by the single supporting point $(x + M - \mathrm{SoC}_v(x), \mathrm{dist}_v(x), 0)$ and SoC_v^x as the function represented by the single supporting point $(x + M - \mathrm{SoC}_v(x), M, 0)$.

4.2 Simplified Vertex Labels

The distance and SoC functions as defined so far contain superfluous information: At a vertex v that can only be reached by recharging at least once, the first interval of the distance function dist_v has the constant value ∞ and the first interval of the SoC function SoC_v has the constant value $-\infty$, signifying that v is not reachable for these amounts of charged energy. Since this interval cannot contain the optimal solution, we can discard it.

A similar situation occurs when we traverse an edge $e = (u, v)$ that causes overcharging to occur for some values of charged energy. In this case, SoC_v has the constant value M in the last interval, while the energy consumption dist_v increases linearly in this interval due to overcharging. Let x be the supporting point of this interval. No value of charged energy above x can lead to the optimal solution, since it leads to a higher total energy consumption without offering a higher SoC. Therefore, we can discard this interval as well, except for x itself.

In summary, we discard all parts of the distance and SoC functions that cause either undercharging or overcharging which can be avoided by charging less. As a result, we are left with only a single *charging interval* $[x_1, x_2]$, where x_1 is the first point at which no undercharging occurs and x_2 is the last point at which no avoidable overcharging occurs. This is exactly the part of the function that was created by charging station, so inside this interval, dist_v is constant and SoC_v increases linearly. We call the size $x_2 - x_1$ of the charging interval the *charging span* of v. Instead of storing an entire function for each vertex, it suffices to store a *vertex label* consisting of the values x_1, x_2, $\text{dist}_v(x_1)$ and $\text{SoC}_v(x_1)$. All other values inside the charging interval can be reconstructed from the values in the label:

$$\begin{aligned}\text{dist}_v(x) &= \text{dist}_v(x_1) + (x - x_1)\\ \text{SoC}_v(x) &= \text{SoC}_v(x_1) + (x - x_1).\end{aligned}$$

When encountering a new charging station, there are now only two supporting points to be considered, and both of them cause the same new supporting point to be added:

$$x_2 + b_{\max} - \text{SoC}_v(x_2) = x_2 + b_{\max} - \text{SoC}_v(x_1) - (x_2 - x_1) = x_1 + b_{\max} - \text{SoC}_v(x_1).$$

This new supporting point simply extends the charging interval beyond x_2 to $x_1 + b_{\max} - \text{SoC}_v(x_1)$, so instead of making a copy of the label, we can simply update x_2. At a battery swapping station, however, we still need make a copy: The original label, which represents the choice of not swapping the battery, remains unchanged. Additionally, we add a second label whose charging interval consists only of the single point $x_1 + b_{\max} - \text{SoC}_v(x_1)$. This label represents the choice of swapping the battery.

In Chapter 3, we established a relationship between the total energy consumption and the amount of charged energy: The total energy consumption is equal to the amount of charged energy plus the difference between the initial SoC at the source vertex s and the current SoC. With the initial SoC at s denoted as $S \in B$, this means for the properties of our vertex labels:

$$\begin{aligned}&\text{dist}_v(x_1) = x_1 + S - \text{SoC}_v(x_1)\\ \Leftrightarrow &x_1 = \text{dist}_v(x_1) + \text{SoC}_v(x_1) - S.\end{aligned}$$

Using this relationship, we can eliminate x_1 from our vertex labels and represent them with only three values in total. We redefine the label of a vertex v as a tuple $\ell_v = (\text{cons}, \text{minSoC}, \text{maxSoC})$, where $\text{cons} = \text{dist}_v(x_1)$ is the amount of energy consumed so far, $\text{minSoC} = \text{SoC}_v(x_1)$ is the lowest SoC with which we can reach v and $\text{maxSoC} = \text{SoC}_v(x_1)$ is the highest SoC with which we can reach v. The following additional properties can be calculated from these values:

- $\text{chargingSpan} = \text{maxSoC} - \text{minSoC}$ is the charging span of the label, i.e., the range of valid SoCs with which v can be reached.
- $\text{minCharged} = \text{cons} + \text{minSoC} - S$ is the lowest amount of charged energy with which v can be reached.
- $\text{maxCharged} = \text{minCharged} + \text{chargingSpan}$ is the highest amount of charged energy with which v can be reached.

We denote the set of all possible labels as $\mathcal{L} := \mathbb{R} \times B \times B$. There are three operations that we can perform on labels: traversing an edge, recharging the battery at a charging station and swapping the battery at a battery swapping station. For the former two operations, we want to take one label and transform it into another label. For the last operation, we want to add a second label while still keeping the original one.

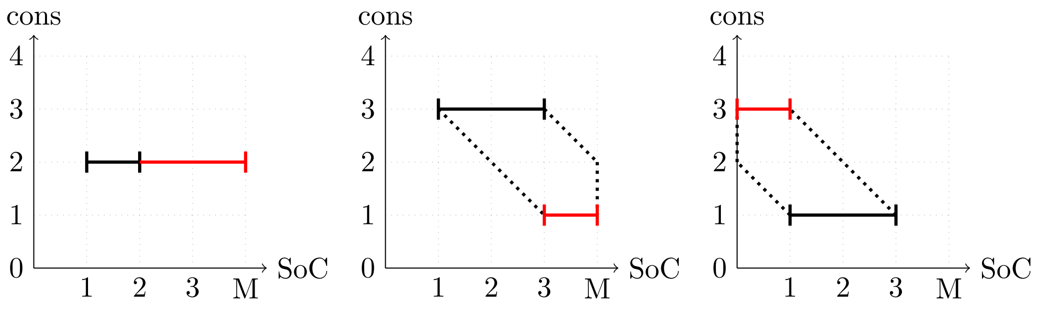

Figure 4.2: Three examples of the link operation for vertex labels. Each label represents an interval of possible SoC values between minSoC and maxSoC, as well a single energy consumption value cons. The left example shows the label $(2, 1, 2)$ encountering a charging station, which leads to a new label $(2, 1, 4)$ with an extended charging interval. The center example shows the label $(3, 1, 3)$ traversing an edge with an energy consumption of -2. The energy consumption is added to cons and subtracted from minSoC and maxSoC, so the label moves along the dotted diagonal lines. In this example, partial overcharging occurs: The maxSoC value of the label reaches the maximum battery capacity $M := 4$, at which point it can no longer increase, cause the charging span to decrease. In the example to the right, the label $(1, 1, 3)$ traverses an edge with an energy consumption of 2. In this case, partial undercharging occurs, limiting minSoC to 0 and decreasing the charging span.

Linking. We combine the two operations of traversing an edge and recharging the battery into a single *link* operation $\circ \colon \mathcal{L} \times (V \cup E) \to \mathcal{L}$ similar to the one introduced in Chapter 2.4.1 for energy consumption functions. When supplied with a label and an edge, it returns the label that results from traversing the edge. When supplied with a label and a vertex, it returns the label that results from recharging the battery at the vertex, provided the vertex contains a charging station. For a label ℓ and an edge e, the link operation is defined as follows:

$$\ell \circ e := \begin{cases} \perp & \text{if } \ell.\operatorname{maxSoC} < \operatorname{cons}(e) \\ ((\ell \circ e).\operatorname{cons}, (\ell \circ e).\operatorname{minSoC}, (\ell \circ e).\operatorname{maxSoC}) & \text{otherwise, with} \end{cases}$$

$$\begin{aligned} (\ell \circ e).\operatorname{cons} &:= \ell.\operatorname{cons} + \max(\operatorname{cons}(e), \ell.\operatorname{minSoC} - M) \\ (\ell \circ e).\operatorname{minSoC} &:= \min(\max(0, \ell.\operatorname{minSoC} - \operatorname{cons}(e)), M) \\ (\ell \circ e).\operatorname{maxSoC} &:= \min(\ell.\operatorname{maxSoC} - \operatorname{cons}(e), M). \end{aligned}$$

Normally, we simply add the edge weight to cons and subtract it from both minSoC and maxSoC. However, if under- or overcharging occurs for part or all of the charging interval, we have to adapt the linked values accordingly:

- If $\ell.\operatorname{maxSoC} < \operatorname{cons}(e)$ holds, the edge cannot be traversed even with the highest possible SoC and the link operation returns $\perp$ to signify this. We call this situation *complete undercharging* because undercharging occurs inside the entire charging interval.
- If $\ell.\operatorname{minSoC} < \operatorname{cons}(e)$ holds, undercharging occurs in at least part of the charging interval. We call this situation *partial undercharging*. Undercharging occurs exactly for those SoC values that drop below 0 after subtracting the edge weight. We cut off that part of the charging interval by setting minSoC to 0.

- If $\ell.\,\text{maxSoC} - \text{cons}(e) > M$ holds, overcharging occurs in at least part of the charging interval. We call this situation *partial overcharging.* Overcharging occurs exactly for those SoC values that rise above M after subtracting the edge weight. As already outlined earlier, this part of the charging interval can never contain the optimal solution, so we cut it off by setting maxSoC to M.
- If $\ell.\,\text{minSoC} - \text{cons}(e) > M$ holds, overcharging occurs for the entire charging interval and thus cannot be avoided. We call this situation *complete overcharging.* The energy consumption of the edge now increases with the SoC at u while the SoC at v is always M. In order to minimize the energy consumption, we pick minSoC as the best available SoC and cut off the rest of the charging interval. This means that both minSoC and maxSoC are M for $\ell \circ e$, while the consumption on e is $\ell.\,\text{minSoC} - M$.

For a label ℓ and a vertex v, the result of the link operation depends on whether the vertex contains a charging station and which type it has. If the vertex contains no charging station, the operation simply returns the original label uncharged. If it contains a regular charging station, maxSoC is set to M. If it contains a supercharger, maxSoC is set to $0.8 \cdot M$ (unless it is already higher than that, in which case the supercharger cannot be used):

$$\ell \circ v := \begin{cases} (\ell.\,\text{cons}, \ell.\,\text{minSoC}, M) & \text{if } v \in \text{CS and type}(v) = \text{REG} \\ (\ell.\,\text{cons}, \ell.\,\text{minSoC}, \max(\ell.\,\text{maxSoC}, 0.8 \cdot M)) & \text{if } v \in \text{CS and type}(v) = \text{SC} \\ \ell & \text{otherwise.} \end{cases}$$

The definition of the link operation can also be extended to paths: A label ℓ is linked with a path $P = (v_1, v_2, \ldots, v_k)$ by linking it with all edges and vertices on the path in the order in which they appear in the path, i.e., first the edge (v_1, v_2), then the vertex v_2, then the edge (v_2, v_3), and so forth. With the set of all possible paths denoted as $\mathcal{P}$, the link operator becomes $\circ\colon \mathcal{L} \times \mathcal{P} \to \mathcal{L}$:

$$\ell \circ (v_1, v_2, \ldots, v_k) := \ell \circ (v_1, v_2) \circ v_2 \circ (v_2, \ldots, v_k).$$

Examples of the link operation for both edges and vertices are shown in Figure 4.2.

Swapping. Battery swapping stations cannot be handled by the link operation because they add a label instead of replacing an existing one. Instead, we define an operation $\text{swap}\colon \mathcal{L} \to \mathcal{L}$ exclusively for battery swapping station. This operation calculates the label that is added by the battery swapping station, which is the original label with minSoC and maxSoC both replaced by M:

$$\text{swap}(\ell) := (\ell.\,\text{cons}, M, M).$$

4.3 Pareto Sets

Before we can adapt our algorithm to use the new vertex labels and operation, we need to solve one remaining problem: One vertex can now have more than one label. We have already seen that the swap operation for battery swapping stations duplicates each label, but there is another reason why a vertex can have multiple labels: Before charging stations were involved, each subpath of a shortest path was also a shortest path itself. Specifically, for a shortest s-t-path $P = (s = v_1, v_2, \ldots, v_k)$, every prefix $P^i = (s = v_1, v_2, \ldots, v_i)$ of P was a shortest s-v_i-path. This meant that in order to find the shortest s-t-path, it was

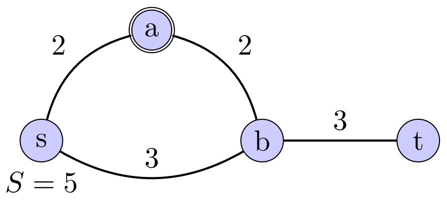

Figure 4.3: In this graph, the vehicle starts at vertex s with an initial SoC of $S = 5$. Vertex a contains a charging station. If we want to reach b, the shortest feasible path is (s, b). However, if we want to reach t, we need to recharge, so the shortest feasible path becomes (s, a, b, t).

sufficient to keep track of the shortest path from s to every vertex v. To see that this is no longer the case once charging stations are involved, we consider the following two examples:

In Figure 4.3, there are two paths from s to b. One uses the direct edge (s, b), the other visits a, which contains a charging station. If we want to reach b from s, the direct edge (s, b) is optimal, since it only consumes 3 energy units while the path over a consumes 4. However, if we want to reach t with an initial SoC of $S = 5$, the path over (s, b) is no longer feasible: The edge (s, b) consumes 3 units, leaving the vehicle with a SoC of 2 at b. This is not sufficient to traverse the edge (b, t), which requires 3 units. If we take the path over a, however, we can recharge the battery at a. Recharging to a SoC of 5 allows us to traverse both (a, b) and (b, t) and thus reach t. We see that while (s, a, b, t) is the shortest feasible s-t-path, its subpath (s, a, b) is not the shortest feasible s-b-path. Since we do not know the further course of the path to t when we reach b, we need maintain labels for both paths. The reason why scenarios like this one can occur is because due to the possibility of recharging at charging stations, the path with the lowest energy consumption is no longer necessarily the one that leaves the vehicle with the highest SoC at its end. Taking a longer path may be necessary if a shorter path results in a SoC that is too low to reach the target. Thus, when deciding which labels to keep, we need to consider not only of the amount of energy consumed so far but also the current SoC.

In Figure 4.4, there are two paths from a to c. One uses the direct edge (a, c), the other visits b. The path over b consumes no energy in total, but it requires a SoC of at least 4 at a to traverse. The direct edge (a, c) requires only a SoC of 1 to traverse but also consumes one energy unit. The vertex a contains a charging station where the vehicle may recharge. If we want to reach c from s with an initial SoC of $S = 5$, the path over b is optimal, since since it consumes only 2 energy units, whereas the path over (a, c) consumes 3. However, we reach a with a SoC of 3, which is not sufficient to traverse (a, b). If we charge one energy unit at a, we are able to traverse (a, b) and reach c with a SoC of 4 and 2 units of consumed energy in total. If we now want to reach t, we must traverse the additional edge (c, t), which allows us to recuperate 3 energy units. However, if the maximum battery capacity is $M = 5$, our current SoC is only one unit below M. Therefore, we can only recuperate one unit, reducing the total amount of consumed energy to 1. If we take the direct edge (a, c) instead of the path over b and do not recharge at a, we reach c with a SoC of 2 and 3 units of consumed energy. When traversing (c, t), we can now recuperate all 3 units of energy, resulting in a SoC of 5 and no energy consumed in total.

We see that recuperation allows a previously longer path to overtake the shortest path if overcharging prevents the formerly shortest path from recuperating all possible energy. Thus, when deciding which labels to keep, we need to consider not only the current total energy consumption, but also lowest possible value that the total energy consumption may reach in the further course of the path. The only way to reduce the total energy consumption is to recuperate energy on an edge with a negative weight, and only so long as the battery

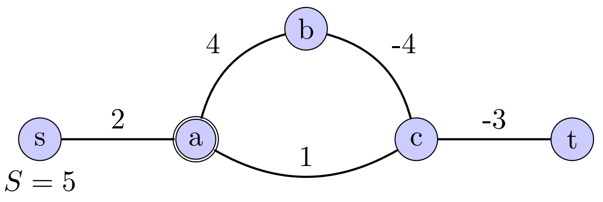

Figure 4.4: In this graph, the vehicle starts at vertex s with an initial SoC of $S = 5$. The maximum battery capacity is $M = 5$. Vertex a contains a charging station. If we want to reach c, the shortest feasible path is (s, a, b, c), which requires recharging at least one energy unit. If we want to reach t, this path leads to overcharging on the edge (c, t), so the shortest feasible path becomes (s, a, c, t), which does not require recharging.

is not already full. Thus, the lowest possible future value of the total energy consumption can be calculated as the current total energy consumption minus the amount of energy that can be recharged before the battery is full. For a given label ℓ this can be calculated as $\ell.\operatorname{cons} - (M - \ell.\operatorname{minSoC})$. Note that $\ell.\operatorname{cons}$ is equal to $S - \ell.\operatorname{minSoC} + \ell.\operatorname{minCharged}$, so the lowest possible energy consumption is $S - M + \ell.\operatorname{minCharged}$. Since S and M are constants, the lowest possible energy consumption depends only on the amount of energy charged so far. Without charging stations, the amount of charged energy is always 0, the lowest possible energy consumption is the same for all labels.

As these two examples demonstrate, it no longer suffices to maintain only one label per vertex v. Instead, we have to maintain one label for each s-v-path that could possibly be a subpath of the shortest s-t-path. Which of these paths is actually a subpath of the shortest s-t-path depends on how the remaining path from v to t progresses.

In order to handle sets of multiple labels per vertex, we introduce a concept that is commonly used in multi-objective optimization problems: the *Pareto set* [Mar84]. In a typical multi-objective optimization problem, we ask for a set of points in a search space $\mathcal{S}$ so that several objective functions $f_1, \dots, f_n \colon \mathcal{S} \to \mathbb{R}$ are minimized. We say that a point $x \in \mathcal{S}$ *dominates* a point $y \in \mathcal{S}$ if $f_i(x) \leq f_i(y)$ holds for all $1 \leq i \leq n$, i.e., if x is equal to or better than y according to all objectives. We call a point $x \in \mathcal{S}$ *Pareto-optimal* if no other point $y \in \mathcal{S}$ exists that dominates it, and we call the set of all Pareto-optimal points the *Pareto set*.

In our case, the search space is the set of all possible vertex labels L. In order to compute a Pareto set that contains exactly those labels that may be part of the optimal solution, we must define a set of suitable objectives. One of these objectives should obviously be minimizing the total energy consumption, since this is our ultimate goal. Additionally, the examples discussed above demonstrate that at least two more objectives are necessary: minimizing the lowest total energy consumption that may be reached in the further course of the path and maximizing the current SoC. We thus define three objective functions $f_1, f_2, f_3 \colon L \to \mathbb{R}$:

$$\begin{aligned} f_1(\ell) &= \ell.\operatorname{cons} \\ f_2(\ell) &= \ell.\operatorname{cons} + \ell.\operatorname{minSoC} \\ f_3(\ell) &= \ell.\operatorname{maxSoC}. \end{aligned}$$

Of these functions, f_1 and f_2 must be minimized, while f_3 must be maximized (which is equivalent to minimizing a function $f_3' \equiv -f_3$). We call the values cons, cons + minSoC

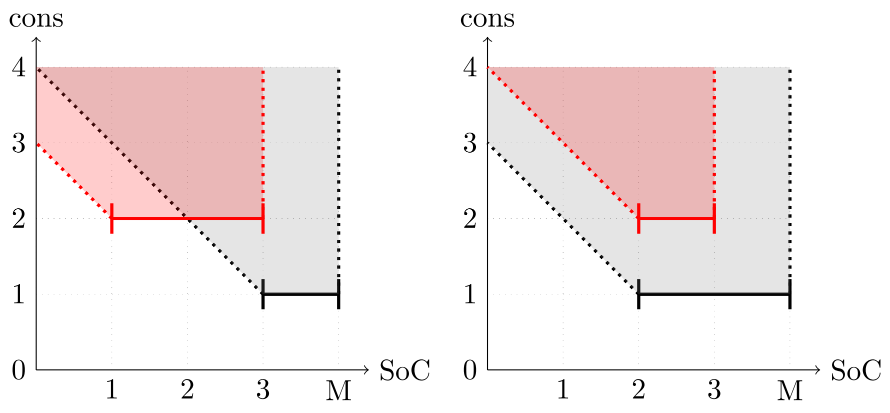

Figure 4.5: A graphical representation of the dominance relation for vertex labels. Each label ℓ dominates the area enclosed by the three inequalities $\ell.\operatorname{cons} \leq \operatorname{cons}$, $\ell.\operatorname{cons} + \ell.\operatorname{minSoC} \leq \operatorname{cons} + \operatorname{SoC}$ and $\ell.\operatorname{maxSoC} \geq \operatorname{SoC}$. Any labels that are located entirely inside this area are dominated by ℓ. In the example to the left, neither of the two labels is located entirely within the area dominated by the other, so neither label is dominated. In the example to the right, however, the red label is located entirely inside the area dominated by the black label and is therefore dominated.

and maxSoC our *dominance criteria*. A label $\ell_1 \in L$ *dominates* a label $\ell_2 \in L$ (denoted by $\ell_1 \propto \ell_2$) if all of the following three conditions hold:

$$\ell_1.\operatorname{cons} \leq \ell_2.\operatorname{cons} \tag{4.1}$$
$$\ell_1.\operatorname{cons} + \ell_1.\operatorname{minSoC} \leq \ell_2.\operatorname{cons} + \ell_2.\operatorname{minSoC} \tag{4.2}$$
$$\ell_1.\operatorname{maxSoC} \geq \ell_2.\operatorname{maxSoC}\,. \tag{4.3}$$

An illustration of the dominance relation is shown in Figure 4.5. Using this definition of dominance, we can adapt our algorithm to handle a whole set of labels L_v at each vertex v. This set contains one label for every Pareto-optimal path from s to v. Whenever an edge $e = (u, v)$ is relaxed, we check if the resulting label ℓ is dominated by any of the labels currently in L_v. If this is not the case, ℓ is added to L_v and any labels in L_v that are dominated by ℓ are removed.

We can extend the link operator to take label sets instead of single labels by performing the link operation on every label in the label set. For a label set L and a path P, for example, it is defined as follows (the definitions for vertices and edges are analogous):

$$L \circ P := \{\ell \circ P \mid \ell \in L\}.$$

In order to model the effects of dominance on our label sets, we define an operation $\operatorname{trim}\colon \mathcal{L} \to \mathcal{L}$ that removes all labels from a label set L that are dominated by another label in L.

Correctness. In order to prove that this strategy always finds the shortest feasible s-t-path, we examine two important properties of our dominance relation:

Lemma 4.1. *The dominance relation $\propto$ as defined above has the following two properties:*

- *Transitivity: If $\ell_1 \propto \ell_2$ and $\ell_2 \propto \ell_3$, then $\ell_1 \propto \ell_3$.*

- *Finality: If $\ell_1 \propto \ell_2$, then* $\text{swap}(\ell_1) \propto \text{swap}(\ell_2)$*, and for any path P, $(\ell_1 \circ P) \propto (\ell_2 \circ P)$.*

Proof. Transitivity follows directly from the definition of dominance, since the relations $\leq$ and $\geq$ are themselves transitive.

Finality for the swap operation is also simple to prove: The swap operation does not change the value of cons but sets both minSoC and maxSoC to M. Therefore, if $\ell_1 \propto \ell_2$, then $\text{swap}(\ell_1) \propto \text{swap}(\ell_2)$ follows. For the link operation, we show that $(\ell_1 \circ v) \propto (\ell_2 \circ v)$ and $(\ell_1 \circ e) \propto (\ell_2 \circ e)$ hold true for any vertex v and any edge e. Since the link operation for a path P consists of consecutive link operations for edges and vertices, $(\ell_1 \circ P) \propto (\ell_2 \circ P)$ then follows from this.

The link operation for vertices always leaves the values of cons and minSoC unchanged. The value of maxSoC depends on the charging station type of v:

- If v does not contain a charging station, then ℓ_1 and ℓ_2 remain entirely unchanged and therefore $(\ell_1 \circ v) = \ell_1 \propto \ell_2 = (\ell_2 \circ v)$ holds true.
- If v contains a regular charging station, then $(\ell_1 \circ v).\text{maxSoC} = (\ell_2 \circ v).\text{maxSoC} = M$ and therefore $(\ell_1 \circ v) \propto (\ell_2 \circ v)$ holds true.
- If v contains a supercharger, then for a label ℓ the value of $\ell.\text{maxSoC}$ is set to $\max(\ell.\text{maxSoC}, 0.8 \cdot M)$. We know that $\ell_1.\text{maxSoC} \geq \ell_2.\text{maxSoC}$, from which it follows that $\max(\ell_1.\text{maxSoC}, 0.8 \cdot M) \geq \max(\ell_2.\text{maxSoC}, 0.8 \cdot M)$. Therefore, $(\ell_1 \circ v) \propto (\ell_2 \circ v)$ holds true.

Proving finality for the link operation on edges is more complicated, since the values of the resulting label depend on whether over- or undercharging occured and on whether the entire charging interval was affected or only part of it. For an edge e and our two labels ℓ_1 and ℓ_1, we consider all possible combinations of situations. We define $\ell_1' := \ell_1 \circ e$ and $\ell_2' := \ell_2 \circ e$. It is impossible for overcharging to occur for one label and undercharging for the other, since the former requires the edge weight to be negative and the latter requires it to be positive. Therefore, we can group the situations by whether overcharging, undercharging or neither of the two occurs for at least one label:

- **Neither over- nor undercharging for both labels:** In this case, the consumption $\text{cons}(e)$ of e is added to cons and subtracted from minSoC and maxSoC for both labels. This means that $\text{cons} + \text{minSoC}$ remains unchanged for both labels, while cons and maxSoC are each shifted by a constant value that is the same for both labels. Therefore, $\ell_1' \propto \ell_2'$ holds true.
- **Undercharging occurs for at least one label:** Whether complete undercharging occurs depends on the value of maxSoC. Since $\ell_1.\text{maxSoC} \geq \ell_2.\text{maxSoC}$, it is impossible for complete undercharging to occur for ℓ_1 but not for ℓ_2. If complete undercharging does occur for ℓ_2, then ℓ_2' is removed from the label set and therefore $\ell_1' \propto \ell_2'$ is trivially fulfilled. In the remaining possible cases, neither complete undercharging nor overcharging occurs for either label. Therefore, the edge weight $\text{cons}(e)$ is simply added to $\ell_1.\text{cons}$ and $\ell_2.\text{cons}$, and therefore $\ell_1'.\text{cons} \leq \ell_2'.\text{cons}$ holds true. Because overcharging does not occur, maxSoC remains unchanged for both labels. The changes to $\text{cons} + \text{minSoC}$ depend on which label experiences partial undercharging:
  - **Partial undercharging occurs for both labels:** In this case, minSoC is set to 0 for both labels, so $\text{cons} + \text{minSoC} = \text{cons}$. Since we know that $\ell_1'.\text{cons} \leq \ell_2'.\text{cons}$, it follows that $\ell_1' \propto \ell_2'$ holds true.

  - **Partial undercharging occurs only for ℓ_1:** In this case, $\ell_1'.\,\mathrm{minSoC}$ is set to 0. Since we know that $\ell_1'.\,\mathrm{cons} \leq \ell_2'.\,\mathrm{cons}$, it follows that $\ell_1'.(\mathrm{cons} + \mathrm{minSoC}) = \ell_1'.\,\mathrm{cons} \leq \ell_2'.\,\mathrm{cons} \leq \ell_2'.(\mathrm{cons} + \mathrm{minSoC})$ holds true, and therefore $\ell_1' \propto \ell_2'$.
  - **Partial undercharging occurs only for** ℓ_2: In this case, $\ell_2'.\,\mathrm{minSoC}$ is set to 0. This means that $\ell_2'.(\mathrm{cons} + \mathrm{minSoC}) = \ell_2.\,\mathrm{cons} + \mathrm{cons}(e) = \ell_2.(\mathrm{cons} + \mathrm{minSoC}) + \mathrm{cons}(e) - \ell_2.\,\mathrm{minSoC}$. Partial undercharging occurs only if $\ell_2.\,\mathrm{minSoC} < \mathrm{cons}(e)$ holds, and we already know that $\ell_1.(\mathrm{cons} + \mathrm{minSoC}) \leq \ell_2.(\mathrm{cons} + \mathrm{minSoC})$. With this, it follows that $\ell_2'.(\mathrm{cons} + \mathrm{minSoC}) > \ell_2.(\mathrm{cons} + \mathrm{minSoC}) \geq \ell_1.(\mathrm{cons} + \mathrm{minSoC}) = \ell_1'.(\mathrm{cons} + \mathrm{minSoC})$. Therefore, $\ell_1' \propto \ell_2'$ holds true.

- **Overcharging occurs for at least one label:** Whether overcharging occurs depends on the value of maxSoC. Since $\ell_1.\,\mathrm{maxSoC} \geq \ell_2.\,\mathrm{maxSoC}$, it is impossible for overcharging (either complete or partial) to occur for ℓ_2 but not for ℓ_1. Thus, at least partial undercharging must occur for ℓ_1. This means that $\ell_1'.\,\mathrm{maxSoC} = M \geq \ell_2'.\,\mathrm{maxSoC}$ automatically holds. We know that $\mathrm{cons} + \mathrm{minSoC}$ is equivalent to $\mathrm{minCharged} + S$, and since the value of minCharged only changes when undercharging occurs, we know that $\mathrm{cons} + \mathrm{minSoC}$ remains unchanged. The changes to cons depend on which label experiences complete overcharging:
  - **Neither label experiences complete overcharging:** This means that $\mathrm{cons}(e)$ is added to cons for both labels. Therefore, $\ell_1' \propto \ell_2'$ holds true.
  - **Complete overcharging occurs for both labels:** This means that cons is set to $\mathrm{cons} + \mathrm{minSoC} - M$ for both labels. Therefore, $\ell_1'.\,\mathrm{cons} = \ell_1.(\mathrm{cons} + \mathrm{minSoC}) - M \leq \ell_2.(\mathrm{cons} + \mathrm{minSoC}) - M = \ell_2'.\,\mathrm{cons}$ and thus $\ell_1' \propto \ell_2'$ holds true.
  - **Complete overcharging occurs only for** ℓ_1: This means that $\ell_1'.\,\mathrm{cons}$ is set to $\ell_1.(\mathrm{cons} + \mathrm{minSoC}) - M$ and $\ell_2'.\mathrm{cons}$ to $\ell_2.\,\mathrm{cons} + \mathrm{cons}(e)$. Since complete overcharging does not occur for ℓ_2, we know that $\ell_2.\,\mathrm{minSoC} \leq M + \mathrm{cons}(e)$ holds. From this it follows that $\ell_1'.\,\mathrm{cons} = \ell_1.(\mathrm{cons} + \mathrm{minSoC}) - M \leq \ell_2.(\mathrm{cons} + \mathrm{minSoC}) - M \leq \ell_2.\,\mathrm{cons} + M + \mathrm{cons}(e) - M = \ell_2.\,\mathrm{cons} + \mathrm{cons}(e) = \ell_2'.\,\mathrm{cons}$. Therefore, $\ell_1' \propto \ell_2'$ holds true.
  - **Complete overcharging occurs only for** ℓ_2: This means that $\ell_1'.\,\mathrm{cons}$ is set to $\ell_1.\,\mathrm{cons} + \mathrm{cons}(e)$ and $\ell_2'.\,\mathrm{cons}$ to $\ell_2.(\mathrm{cons} + \mathrm{minSoC}) - M$. Since complete overcharging occurs for ℓ_2, we know that $\ell_2.\,\mathrm{minSoC} > M + \mathrm{cons}(e)$ holds. From this it follows that $\ell_2'.\,\mathrm{cons} = \ell_2.(\mathrm{cons} + \mathrm{minSoC}) - M > \ell_2.\,\mathrm{cons} + M + \mathrm{cons}(e) - M = \ell_2.\,\mathrm{cons} + \mathrm{cons}(e) \geq \ell_1.\,\mathrm{cons} + \mathrm{cons}(e) = \ell_1'.\,\mathrm{cons}$. Therefore, $\ell_1' \propto \ell_2'$ holds true.

In summary, we see that $(\ell_1 \circ e) \propto (\ell_2 \circ e)$ and $(\ell_1 \circ v) \propto (\ell_2 \circ v)$ hold for any edge e and any vertex v. Thus, $(\ell_1 \circ P) \propto (\ell_2 \circ P)$ also holds for any path P. □

Together, these two properties mean that once a label ℓ is dominated by another label ℓ', we can safely delete it from its label set. For any further path P, finality ensures that $\ell \circ P$ will continue to be dominated by $\ell' \circ P$, while transitivity ensures that even if $\ell' \circ P$ is itself dominated by another label ℓ'', $\ell \circ P$ will still be dominated by ℓ''. Thus, every further label set will contain at least one label that dominates ℓ. Using this, it is easy to show that our strategy always finds the shortest feasible s-t-path:

Theorem 4.2. *Given a graph $G = (V, E)$ with source and target vertices $s, t \in V$, the label set L of t always contains a label ℓ that corresponds to a shortest s-t-path P.*

Proof. We prove this claim by contradiction: Assume that $\ell \notin L$. This means that there is another label $\ell' \in L$ that dominates ℓ. This, however, means that $\ell'.\,\mathrm{cons} \leq \ell.\,\mathrm{cons}$, so ℓ' itself corresponds to a shortest s-t-path, which contradicts our assumption. □

Algorithm 4.1: Charging Label Algorithm

```
Input: Graph $G = (V, E)$, energy consumption function cons, source and target
       vertices $s, t$, range of valid battery states $B = [0, M]$, initial SoC $b(s)$,
       charging stations CS $\subseteq V$
Data: Priority queue Q
Output: Array labels[$\cdot$] containing a label set for each vertex in V
// Initialization
1  forall $v \in V$ do
2      labels[$v$] $\leftarrow \emptyset$
3  labels[$s$] $\leftarrow \{(0, S, S)\}$
4  Q.Insert($s$, Key(labels[$s$]))
// Main loop
5  while not Q.IsEmpty() do
6      $u \leftarrow$ Q.DeleteMin()
7      $\ell \leftarrow$ Settle(labels[$u$])
8      forall $e = (u, v) \in E$ do
9          $\ell' \leftarrow \ell \circ e \circ v$
10         $L \leftarrow \{\ell'\}$
11         if $v \in$ CS and type($v$) = BSS then
12             $L \leftarrow L \cup \{$swap$(\ell')\}$
13         if Add(labels[$v$], $L$) then
14             if Q.Contains($v$) then
15                 Q.DecreaseKey($v$, Key(labels[$v$]))
16             else
17                 Q.Insert($v$, Key(labels[$v$]))
18     if HasUnsettledLabels(labels[$u$])) then
19         Q.Insert($u$, Key(labels[$u$])
```

4.4 Basic Approach

By using Pareto sets at each vertex to represent all labels that might lead to the optimal solution, we can extend our algorithm to solve the EEVRC problem. The resulting algorithm is a variation of the Multi-Objective Shortest Path Search algorithm introduced by Martins [Mar84]. Because the labels of our algorithm use intervals to represent each valid amount of charged energy, we call our algorithm the Charging Label Algorithm (CLA). The pseudocode for this algorithm is shown in Algorithm 4.1.

Instead of two arrays dist[$\cdot$] and SoC[$\cdot$] which hold the tentative energy consumption dist[v] and SoC SoC[v] for each vertex v, the algorithm now stores a set of labels for each vertex v. Each label represents one tentative Pareto-optimal s-v-path and consists of a tuple of three values (cons, minSoC, maxSoC), which represent the energy consumption and the minimal and maximal SoC with which v can be reached, respectively.

As before, we use a priority queue Q that maintains visited but unsettled labels during the main loop of the algorithm. In Dijkstra's algorithm, the key for the priority queue was the tentative distance from s, which caused the labels to be settled in increasing order of consumed energy. We still want this to be the case, but it is now possible for two labels to have the same cons value if they differ in the other two values. To prevent ambiguities, we sort the labels according to their *lexicographical order*, with cons as the first criterion, minSoC as the second and maxSoC as the third. This way, the labels are still sorted in increased order of consumed energy, but the other two label values are used to break ties if necessary.

The label sets are stored in an array labels[$\cdot$] that contains one entry labels[v] for every vertex v. Each label set itself is an array in which the labels are sorted according to their key, which is the order in which they will be settled by the algorithm. Additionally, the label set maintains a pointer to its first unsettled label. Whenever we visit a new vertex v, we insert its label set labels[v] into Q, using the key of its first unsettled label. In each iteration of the main loop, we extract the label set with the lowest key from the queue. Then we extract its first unsettled label and settle it. The label sets must provide the following operations:

- `Key(`L`)` returns the key of the first unsettled label in L. This is used as the key for L itself in Q.
- `Settle(`L`)` returns the first unsettled label of L and marks it as settled, which is done by advancing the pointer to the first unsettled label in L to the next label.
- `HasUnsettledLabels(`L`)` returns a boolean value, which is `true` if and only if L contains at least one unsettled label. This is the case if the pointer to the first unsettled still points to an actual label in L. If L contains no more unsettled labels, the pointer points behind the end of the array in which the labels are stored.
- `Add(`L, L'`)` adds the labels in L' to L while removing dominated labels. For every label $\ell \in L'$, the operation checks if ℓ is dominated by an existing label in L, and if so, discards it. Otherwise, ℓ is inserted into L so that L remains sorted according to key. All labels in L that are dominated by ℓ are removed. If necessary, the pointer to the first unsettled label is adjusted. The operation returns a boolean value, which is `true` if and only if at least one label in L' was successfully added to L.

During the initialization phase, the label sets of all vertices are initialized as empty sets, to signify that no paths have been found yet. The label set labels[s] is initialized with the initial label $(0, S, S)$, where $S \in B$ is the initial SoC. Then, they key of labels[s] is calculated and s is inserted into Q.

In each step of the main loop, we extract the vertex u from Q whose label set labels[u] has the smallest key. Then we settle the first unsettled label ℓ from labels[u]. Because we used the key of ℓ as the key of labels[u] in Q, we know that this is the label with the lowest energy consumption among all unsettled labels currently in Q. After settling ℓ, we scan all outgoing edges of u. We relax each edge $e = (u, v)$ by linking ℓ with e and v to obtain a new label ℓ' for v. If v contains a battery swapping station, we also calculate the additional label swap(ℓ') that represents swapping the battery at v.

We try to add ℓ' (and possibly swap(ℓ') as well) to the label set of v and remove all labels that become dominated. If at least one of the two labels was successfully, we recalculate the key of labels[v] and update Q accordingly. If the label set of u still has unsettled labels after settling ℓ, we reinsert it into Q as well, using the key of its next unsettled label.

Label-Correcting Algorithm. Due to the existence of negative edge weights, it is possible that a label is added to a label set whose key is lower than that of the first unsettled label. It is also possible that a newly added label dominates already settled labels, which means they were settled unnecessarily. Because of this, we call the algorithm *label-correcting.* If all edge weights are non-negative, it is guaranteed that an already settled label is not dominated later, and thus we call the algorithm *label-setting.*

Path Retrieval. Once the algorithm has terminated, the first label in the label set labels[t] of t represents the shortest feasible s-t-path P, since it is the one with the lowest energy consumption. In order to reconstruct P, we extend the labels by parent pointers that point both the previous vertex in P and the label at that vertex from which the current label originates. By following these parent pointers until s is reached, P can be reconstructed. However, in order to completely solve the EEVRC problem, we also need to determine the charging distribution function $\mathrm{cd}\colon\ \mathrm{CS} \cap P \to B$, which tells us where to charge and how much.

The problem statement requires that the number of used charging stations is minimized. We can achieve this by iterating through P, keeping track of the current SoC and the last visited charging station. Whenever the current SoC drops below 0, we go back to the last visited charging station and charge as much as possible there. This will give us enough energy to continue, because otherwise P would not be feasible, which we know it is. This strategy minimizes the number of used charging stations because it charges only when it is necessary to continue, and then as much as possible. Charging when it is not necessary to continue would unnecessarily increase the number of used charging stations, while charging less than is possible at a used charging station would result in a lower SoC, which would decrease the distance the vehicle could travel before having to charge again.

However, one problem remains: Determining how much can be charged at a given charging station cs is not trivial. We decide to charge at cs whenever the SoC at the current vertex v drops below. We could charge just enough energy to reach the maxSoC value of the label at v. This would maximize the SoC at v and prevent overcharging between cs and v, but overcharging might still occur after v, for example if the next edge has a negative weight. To avoid this, we must delay the decision and iterate further across P until we reach a point where the amount of energy that was charged at cs no longer has an influence on the current SoC. This is the case once we use another charging station cs′. After cs′, the current SoC only depends on the amount of energy we charged at cs′ and not on the amount of energy we charged at cs.

To decide how much to charge at cs, we need to know the value of maxSoC when cs′ is reached, but before it is used. However, the way the algorithm works, the label at cs′ only contains the maxSoC value after using cs′. To retrieve the value we need, we can either go back to the previous vertex and re-traverse the edge that led to cs′, or we can adapt the algorithm so that whenever it links a label with a charging station, it makes a copy of the label beforehand.

In summary, the charging distribution cd for the shortest feasible path P can be found as follows: We iterate through P, keeping track of the current SoC as well as the last visited and last used charging station. Whenever the SoC at the current vertex v drops below 0, we go back to the last visited charging station cs and set the preliminary value of cd(cs) so that the vehicle reaches v with the maxSoC value of the label at v. If cs is a supercharger, it is possible that it does not offer enough energy for this. In this case, there must be another charging station before cs that does offer enough energy, so we charge there instead. After setting cd(cs), we mark cs as used. Then we go back to the last used charging station cs′ before cs. We set the permanent value of cd(cs′) so that the vehicle reaches cs′ with the maxSoC value of the label at cs′ before recharging. We also do this once we reach t.

The worst-case complexity of this procedure is linear in the size of P, so compared to the running time of the main algorithm it is likely to be fast. On realistic graphs, overcharging is unlikely to occur after a charging station, since it is not practical to place a charging station in a location where energy can be recuperated directly after recharging the battery.

Therefore, our algorithm will usually charge the battery to full capacity at every used charging station (and 80% of the full capacity at superchargers).

Complexity. We analyze the complexity of our algorithm under the condition that it is label-setting. This is only the case if the graph does not contain negative edges, so in the next section we will adapt the potential shifting technique to ensure this. Unlike Dijkstra's algorithm, which settled every vertex and relaxed each edge only once, our algorithm may settle vertices and relax edges multiple times, once for each label in the label set of the vertex. The complexity of the algorithm thus depends on the size of the label sets, which we will analyze in Chapter 5. For now, we denote the maximum size of any label set in G after the algorithm has terminated by $|L|$ and analyze the complexity of our algorithm depending on the value of $|L|$.

As before, the initialization phase takes $\mathcal{O}(n)$ time. In each iteration of the main loop, one label is settled. Due to the label-setting property, we know that after the algorithm has terminated, the label set of a vertex v consists exactly of those labels that were settled by the algorithm. The number of labels that are settled per vertex is thus at most $|L|$ and the main loop is executed at most $n|L|$ times. Each iteration of the main loop calls the operations `IsEmpty`$()$, `DeleteMin`$()$, `Settle`$(\cdot)$, `HasUnsettledLabels`$(\cdot)$, `Insert`$(\cdot,\cdot)$ and `Key`$(\cdot)$ at most once each. Additionally, `Insert`$(\cdot,\cdot)$ is called every time a vertex is settled.

Each edge $e = (u, v)$ is relaxed at most once for every label in the label set of u, so the inner loop of the algorithm is executed at most $m|L|$ times. Each iteration of the main loop calls the operation $\circ$ twice and the operations swap$(\cdot)$, `Add`$(\cdot,\cdot)$, `Contains`$(\cdot)$ and `DecreaseKey`$(\cdot,\cdot)$ at most once.

The operations `Key`$(\cdot)$, `Settle`$(\cdot)$ and `HasUnsettledLabels`$(\cdot)$ of a label set can all be implemented in constant time. The complexity of `Add`$(\cdot,\cdot)$ depends on how many labels are added. In our algorithm, these are never more than two. Each added label has to be compared with every other label in the label set in order to determine which labels are dominated. Therefore, `Add`$(\cdot,\cdot)$ has a worst-case complexity of $\mathcal{O}(|L|)$. Altogether this yields a running time of

$$\mathcal{O}(|L|(n(1 + T_{\texttt{IsEmpty}} + T_{\texttt{DeleteMin}} + T_{\texttt{Insert}}) + m(1 + |L| + T_{\texttt{Contains}} + T_{\texttt{DecreaseKey}})).$$

With a Fibonacci heap, this yields a worst-case complexity of $\mathcal{O}(n|L|\log n + m|L|^2)$. With a binary heap it yields $\mathcal{O}((n + m)|L|\log n + m|L|^2)$.

4.5 Speedup Techniques

We have now established an algorithm that solves the EEVRC problem with a worst-case complexity that is polynomial in the size of the graph and the size of the label sets if the graph contains no negative edges. However, this is not the case in our scenario, so the complexity may be exponential. In order to improve the running time of our algorithm, we examine several speedup techniques.

Pruning. So far our algorithm does not have a stopping criterion, so it only terminates once every edge that is reachable from s is relaxed. If the algorithm were label-setting, we could stop once we reach t for the first time, because we would know that first label ℓ that is added to t will have a lower cons value than all other labels that might be added later on. However, due to negative weights, this is not the case in our scenario. The shortest path may cause the energy consumption to rise above $\ell.\text{cons}$ momentarily but then drop below it again due to recuperation.

Because our graph contains no negative cycles, there is a limit to how much energy the vehicle can recuperate when starting at a given vertex v. Energy can only be recuperated on paths with a negative total energy consumption, so the maximum amount of energy that can be recuperated from v is given by the path with the lowest energy consumption starting from v (or 0 if all of these paths have a positive energy consumption). Under our energy consumption metric, this is the shortest path starting from v.

To calculate the maximum recuperation for a vertex v, we could simply perform a Dijkstra query from v and return the lowest energy consumption value that was seen at any settled vertex. If we want to calculate the maximum recuperation for every vertex in G, however, the following approach is faster: We set the energy consumption for every vertex to 0 and initialize the priority queue by inserting all vertices at once. Then we perform a backwards search. Because the energy consumption values are initialized with 0, paths whose energy consumption is positive will automatically be discarded. Once the algorithm has terminated, the energy consumption value at any given vertex v represents the maximum recuperation that is possible when starting from v.

We use this query to precalculate an array $\mathrm{recup}[\cdot]$, which holds the maximum recuperation $\mathrm{recup}[v]$ for every vertex v. Using this information, we can prune our main algorithm: We know that the cons value of the first label ℓ_t in the label set $\mathrm{labels}[t]$ of t is an upper bound for the length of the shortest feasible s-t-path. Every time we relax an edge $e = (u, v)$ to obtain a new label ℓ for v, we check if $\ell.\mathrm{cons} - \mathrm{recup}[v] > \ell_t.\mathrm{cons}$ holds. If this is the case, we can discard the label, because even if it recuperates the maximum possible amount of energy, its energy consumption will still be higher than that of ℓ_t.

While this pruning technique still does not give us a stopping criterion, it allows us to discard labels once their energy consumption reaches too high above that of the shortest path found so far. Because paths with negative energy consumption are fairly short in realistic graphs, this will usually cause the algorithm to terminate shortly after the shortest feasible path has been found.

Improved Dominance Criteria. We can also use the information about maximum recuperation to improve our dominance criteria for the label sets. As outlined in Section 4.3, one of our dominance criteria is the lowest possible total energy consumption that can be reached in the further course of the path. It can be calculated as the current total energy consumption minus the amount of energy that can be recharged before the battery is full. For a label ℓ at a vertex v, this value is given by $\ell.\mathrm{cons} - (M - \ell.\mathrm{minSoC})$. We removed the constant value M to obtain the dominance criterion $\ell.\mathrm{cons} + \ell.\mathrm{minSoC}$.

By incorporating the information about maximum recuperation from each vertex, we can make this dominance criterion even stricter: The maximum amount of energy that can be recuperated when starting from v is $\mathrm{recup}[v]$, so the lowest possible total energy consumption cannot be lower than $\ell.\mathrm{cons} - \mathrm{recup}[v]$. It may be higher than this value if $(M - \ell.\mathrm{minSoC}) < \mathrm{recup}[v]$ holds, which means that the battery becomes full before all energy can be recharged. In this case the lowest total energy consumption is still given by $\ell.\mathrm{cons} - (M - \ell.\mathrm{minSoC})$. By combining these two lower bounds, our new value for the lowest possible energy consumption becomes $\ell.\mathrm{cons} - \min(M - \ell.\mathrm{minSoC}, \mathrm{recup}[v])$.

We replace the dominance criterion $\mathrm{cons} + \mathrm{minSoC}$ with this new, stricter criterion, obtaining the following definition of dominance: A label ℓ_1 at a vertex v_1 dominates a label ℓ_2 at a vertex v_2 if all of the following three conditions hold:

$$\begin{gathered}
\ell_1.\,\mathrm{cons} \leq \ell_2.\,\mathrm{cons} \\
\ell_1.\,\mathrm{cons} - \min(M - \ell_1.\,\mathrm{minSoC}, \mathrm{recup}[v_1]) \leq \ell_2.\,\mathrm{cons} - \min(M - \ell_2.\,\mathrm{minSoC}, \mathrm{recup}[v_2]) \\
\ell_1.\,\mathrm{maxSoC} \geq \ell_2.\,\mathrm{maxSoC}\,.
\end{gathered}$$

Potential Shifting. In order to make our algorithm label-setting, we use the potential shifting technique for electric vehicles as discussed in Chapter 2.4.2. We extend our labels by a fourth value shiftedCons, which holds the energy consumption under the shifted energy consumption functions c'_e. However, we also retain the original energy consumption value cons in our labels, so when performing the link operation, we have to update both values. To ensure that the labels are settled in increasing order of shifted energy consumption, we modify the operation `Key`$(\cdot)$: Instead of using cons as the primary values for the lexicographical ordering, we use shiftedCons. The other values remain unchanged. We only use shiftedCons for sorting the labels; for all other purposes, especially for determining dominance, we still use the original energy consumption value cons.

Because the algorithm is now label-setting, we can terminate the algorithm once we reach t for the first time. This stopping criterion eliminates the need for the pruning technique developed earlier, but we can still use the improved dominance criteria.

5. Complexity Analysis for Charging Label Algorithm

In the previous chapter, we introduced the Charging Label Algorithm (CLA), a variation of the multi-objective shortest path search algorithm that solves the EEVRC problem. We observed that the complexity of this algorithm depends on the size of the Pareto sets that hold the labels for each vertex. In general, the size of the Pareto sets used in a multi-objective shortest path search algorithm may be exponential in the size of the graph. In this chapter, we will analyze the size of the Pareto sets for our algorithm and introduce a class of graphs called *chain graphs* in which the Pareto sets have polynomial size, yielding a polynomial running time for our algorithm on these graphs.

5.1 Exponential Label Set Size for Multi-Objective Dijkstra

A common application for a multi-objective version of Dijkstra's algorithm is the Constrained Shortest Path (CSP) problem, which is $\mathcal{NP}$-complete [GJ79]. In addition to an edge weight $\omega\colon E \to \mathbb{R}$, this problem also features a resource consumption function $\mathrm{rc}\colon E \to \mathbb{R}_{\geq 0}$. The objective is to find a shortest path P under the edge weight ω whose total resource consumption $\mathrm{rc}(P)$ does not exceed an upper bound $R \in \mathbb{R}_{\geq 0}$. The resource consumption function is similar to the energy consumption function in our problem, except that the resource consumption may not be negative, i.e., recuperation is impossible. In this section we will examine a version of the CSP problem where the edge weight is the driving time $\mathrm{dt}\colon E \to \mathbb{R}$. The objective is to find a path that minimizes the driving time while not exceeding the upper bound R for the resource consumption.

This problem can be solved with a multi-objective version of Dijkstra's algorithm where the vertex labels are tuples $(\mathrm{dt}, \mathrm{rc})$, where dt is the total driving time so far and rc is the total resource consumption so far. A label is linked with an edge by simply adding the driving time and resource consumption to the values in the label. If the resource consumption of a label exceeds R, it is discarded.

Consider two labels $\ell_1 = (\mathrm{dt}_1, \mathrm{rc}_1)$ and $\ell_2 = (\mathrm{dt}_2, \mathrm{rc}_2)$ with $\mathrm{dt}_1 < \mathrm{dt}_2$ and $\mathrm{rc}_1 > \mathrm{rc}_2$. Normally, ℓ_1 will be superior to ℓ_2 because its driving time is lower. However, its resource consumption is higher. In the further course of the path, rc_1 may exceed R, causing ℓ_1 to be discarded. In this case, ℓ_2 will be optimal. Because both labels may be optimal depending on the further course of the path, we must maintain a Pareto set of labels for each vertex. In order to use Pareto sets, we must define a dominance relation. We say

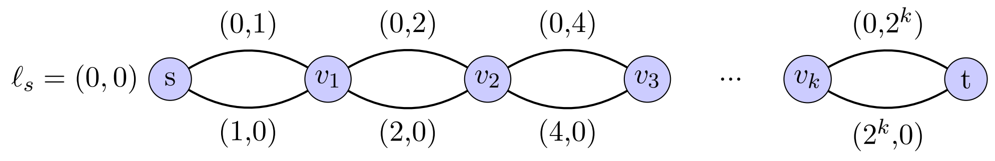

Figure 5.1: An example of a graph that leads to Pareto sets of exponential size. The vehicle starts at s with the initial label $\ell_s = (0,0)$. Each new vertex v_i $(1 < i \leq k+1)$ can be reached via two edges, so for each label in the Pareto set of v_{i-1} there are two in the Pareto set of v_i. None of these labels dominate each other, so after k vertices, the Pareto set contains 2^k labels.

that a label ℓ_1 dominates another label ℓ_2 (denoted by $\ell_1 \propto \ell_2$) if both of the following conditions hold:

$$\ell_1.\,\mathrm{dt} \leq \ell_2.\,\mathrm{dt}$$
$$\ell_1.\,\mathrm{rc} \leq \ell_2.\,\mathrm{rc}\,.$$

Defined as such, the label sets may have exponential size in the size of the graph. Consider for example the graph shown in Figure 5.1. This graph consists of a sequence $(s = v_0, v_1, \ldots, v_k, v_{k+1} = t)$ of vertices. From each vertex v_i $(0 \leq i \leq k)$, there are two edges leading to the next vertex v_{i+1} in the sequence: one with the weight $(0, 2^i)$ and one with the weight $(2^i, 0)$. If we perform an s-t-query on this graph with the initial label $\ell_s = (0,0)$, the size of the Pareto set will double at each vertex, eventually reaching a size of 2^k labels at t. At each vertex v_i, there are two edges leading to the next vertex v_{i+1}, so for each label in the Pareto set of v_i, there will be two at v_{i+1}. Neither of these two labels dominates the other one, since one edge is superior to the dominance criterion dt and the other is superior according to rc.

Eventually, the vehicle reaches t with the Pareto set $L = \{(i, 2^k - i) \mid 0 \leq i \leq 2^k\}$, provided that $R \geq 2^n$ holds. None of these labels dominate each other: Consider two labels $(i, 2^k - i), (j, 2^k - j) \in L$. Without loss of generality, assume that $i < j$. This implies $2^k - i > 2^k - j$ and therefore neither of the two labels dominates the other. There are 2^k labels in L, compared to $k+2$ vertices and $2(k+1)$ edges in the graph. Thus, the size of L is exponential in the size of the graph.

Note that in this version of the CSP problem, the two label properties dt and rc are independent of each other and each one directly corresponds to one edge weight and one dominance criterion. This allows us to create edges that manipulate only one dominance criterion without influencing the other, which was exploited in the example graph to avoid dominance and create exponentially many labels. Returning to the EEVRC problem and CLA, we observe that this is no longer the case. There is only one edge weight cons, which affects the values of all three label properties and thus all three dominance criteria. This means that manipulating one dominance criterion may also inadvertently influence the others.

In the following sections, we generalize the graph from Figure 5.1 to a class of graphs called *chain graphs*. These graphs have the same basic structure as our example graph, consisting of a sequence of vertices connected by multiple concurrent paths. For these graphs, we can show that the size of the label sets cannot become exponential because a sufficient number of labels is always deleted due to dominance.

5.2 Irrelevance of Dominance Criteria

Before we analyze the characteristic structure of chain graphs, we must be able to identify situations in which labels are deleted due to dominance. For this purpose we introduce the concept of the *irrelevance* of dominance criteria. We call a dominance criterion irrelevant among a label set if the dominance relation between any two labels in the set remains exactly the same if we omit the criterion from our definition of dominance and evaluate only the other two criteria. Because our dominance criteria take the form of simple inequations between label properties, there is always a label in every label set that is optimal according to a specific dominance criterion. This means that if two out of the three dominance criteria become irrelevant, the label that is optimal according to the remaining criterion will dominate all the other ones and thus all but one label in the set will be deleted.

Before we can establish a definition of irrelevance, we need to identify the conditions that allow us to omit a criterion from the definition of dominance:

Definition 5.1. *Let $L \subseteq \mathcal{L}$ be a set of labels and* crit *and* crit′ *be two different dominance criteria (*$\mathrm{crit} \neq \mathrm{crit}' \in \{\mathrm{cons}, \mathrm{cons} + \mathrm{minSoC}, -\mathrm{maxSoC}\}$*).*

- *We call L* crit*-equal if $\forall \ell \in L : \ell.\mathrm{crit} = c$ for a constant value $c \in \mathbb{R}$.*
- *We call* crit *and* crit′ *equivalent in L if $\forall \ell \in L : \ell.\mathrm{crit} - \ell.\mathrm{crit}' = c$ for a constant value $c \in \mathbb{R}$.*

Both of these conditions allow us to omit a dominance criterion: If a label set L is crit-equal, then $\ell_1.\mathrm{crit} \leq \ell_2.\mathrm{crit}$ is always fulfilled for any two labels $\ell_1, \ell_2 \in L$, so we no longer have to evaluate crit at all. If two criteria crit and crit′ are equivalent, they always evaluate to the same result for any two labels $\ell_1, \ell_2 \in L$: $\ell_1.\mathrm{crit} \leq \ell_2.\mathrm{crit} \Leftrightarrow \ell_1.\mathrm{crit}' \leq \ell_2.\mathrm{crit}'$. This means that we only need to evaluate one of the two criteria and can omit the other.

With these two conditions, we can give a definition of irrelevance:

Definition 5.2. *Let $G = (V, E)$ be a graph, $L \subseteq \mathcal{L}$ a set of labels at a vertex $v \in V$,* $\mathrm{crit} \in \{\mathrm{cons}, \mathrm{cons} + \mathrm{minSoC}, \mathrm{maxSoC}\}$ *a dominance criterion and $P = (s, v_1, v_2, \dots, v_k)$ a path in G.*

- *We call* crit *irrelevant in L if L is* crit*-equal or if* crit *is equivalent to another dominance criterion* $\mathrm{crit}' \in \{\mathrm{cons}, \mathrm{cons} + \mathrm{minSoC}, \mathrm{maxSoC}\}$ *(*$\mathrm{crit} \neq \mathrm{crit}'$*) in L.*
- *We call P* crit*-omitting if* crit *is not irrelevant in L but is irrelevant in $L \circ P$.*

Before we analyze under which circumstances the link and charge operations in our algorithm allow us to omit dominance criteria, we observe that cons and cons + minSoC become equivalent among a label set if minSoC has the same value for all labels in the set:

Lemma 5.3. cons *and* cons + minSoC *become equivalent among a set of labels L if $\forall \ell \in L : \ell.\mathrm{minSoC} = c$ for a constant value $c \in \mathbb{R}$.*

Proof. Assume that there exists a constant value c so that $\forall \ell \in L : \ell.\mathrm{minSoC} = c$. Then we observe:

$$\forall \ell \in L : (\ell.\mathrm{cons} + \ell.\mathrm{minSoC}) - \ell.\mathrm{cons} = \ell.\mathrm{minSoC} = c.$$

Because c is constant, cons and cons + minSoC fulfill the definition of equivalence. □

With this we can examine the available operations, starting with charging stations:

- *Regular charging station*: A regular charging station sets maxSoC $= M$ for all labels and is therefore maxSoC-omitting in the entire label set.
- *Supercharger*: A supercharger does not affect labels with maxSoC $> 0.8 \cdot M$. For all other labels, the supercharger sets maxSoC $= 0.8 \cdot M$ and is therefore maxSoC-omitting among them.
- *Battery swapping station*: A battery swapping station duplicates every label in the label set. Among the duplicated labels, minSoC and maxSoC are both set to M. Per Lemma 5.3, this means that cons and cons + minSoC become equivalent, so we can omit cons + minSoC. Since maxSoC $= M$, we can also omit maxSoC, so cons remains as the only relevant criterion among the newly added labels. This means that the label with the smallest cons value dominates all the other ones, and thus a battery swapping station can add at most one label to the label set.

When traversing an edge, the result depends on whether over- or undercharging occurs. If neither occurs, traversing an edge simply adds the edge weight to cons and subtracts it from minSoC and maxSoC. cons + minSoC is therefore not changed at all. Shifting a dominance criterion by the same constant value for all labels has no effect on dominance, since the shifted criterion is equivalent to the original criterion. Therefore, no dominance criterion is omitted in this case. However, over- and undercharging have additional effects on the labels. We can divide the label set into five groups according to whether they are affected by over- or undercharging:

- *Not affected by over- or undercharging*: As observed above, no dominance criterion becomes irrelevant.
- *Affected by complete undercharging*: These labels become infeasible and are removed from the label set.
- *Affected by partial undercharging*: Partial undercharging sets minSoC $= 0$ among all affected labels, so per Lemma 5.3 cons and cons + minSoC become equivalent and we can omit cons + minSoC among this group.
- *Affected by partial overcharging*: Partial overcharging sets maxSoC $= M$ among all affected labels and we can thus omit maxSoC among this group.
- *Affected by complete overcharging*: Complete overcharging sets minSoC $= M$ and maxSoC $= M$. This causes both cons + minSoC and maxSoC to become irrelevant. cons remains as the only relevant criterion, and thus one label dominates all others.

Note that undercharging can only occur when the edge weight is positive and overcharging can only occur when the edge weight is negative, so overcharging and undercharging never occur on the same edge. However, both can occur on a single path.

In total, we observe that the only operation that does not omit any criterion is traversing an edge without over- or undercharging. Regular charging stations, superchargers and partial overcharging omit maxSoC, partial undercharging omits cons + minSoC, and battery swapping stations and complete overcharging omit both cons + minSoC and maxSoC. However, these operations may omit their respective criteria only on part of the label set.

The only operation that adds labels is a battery swapping station. Aside from that, the only situation in which the size of the label set can increase is when two edges lead to the same vertex, causing two label sets to be merged. In the following, we assume that the graph contains no battery swapping stations or superchargers, only regular charging stations.

Now that we know which operations cause which criteria to become irrelevant, we can try to partition label sets into different groups, depending on which operations they encountered.

If done correctly, this allows us to identify groups among which one or more dominance criteria are irrelevant. To aid us in finding these groups, we make the following observations:

Lemma 5.4. *Let $P = (s = v_1, v_2, \dots, v_k = t)$ be a path and L_s the label set at s. The following is true of the label set $L_t \coloneqq \text{trim}(L_s \circ P)$ at t:*

1. cons + minSoC *is irrelevant among all labels that have encountered partial undercharging in P.*
2. *There is at most one label in L_t that has encountered complete overcharging in P.*
3. maxSoC *is irrelevant among all labels in L_t that have encountered overcharging in P.*
4. *If P contains at least one charging station,* maxSoC *is irrelevant among L_t.*

Proof.

1. Directly after encountering partial undercharging, minSoC is 0 among all affected labels, so per Lemma 5.3 cons + minSoC is irrelevant among them. Let B be the set of labels that have already been affected by undercharging. Because minSoC cannot drop below 0, no label has a lower minSoC value than the ones in B. This remains true in the further course of P, regardless of which operations are performed: Charging does not affect minSoC at all while traversing an edge without complete overcharging or partial undercharging shifts minSoC by the same value for all labels. If partial undercharging occurs again, it affects all labels whose minSoC value is smaller than the weight of the edge. Because the labels in B have the lowest minSoC value among the label set, they will be affected again now. Further labels may be added to B, but since minSoC is set to 0 among both the old labels and the new labels in B, cons + minSoC is still irrelevant among all of B. Since all labels in B have the same minSoC value, complete overcharging will affect either all or none of B. If it affects all of B, minSoC will be set to M, so cons + minSoC will still be irrelevant.
2. Complete overcharging sets both minSoC and maxSoC to M, which causes both cons + minSoC and maxSoC to become irrelevant. Therefore, once complete overcharging occurs on a set of labels, only one label ℓ remains among them. ℓ now has the highest minSoC value in the label set, so if complete overcharging occurs again and affects other labels, ℓ will also be affected, so again only one label will remain among them.
3. Directly after encountering overcharging, maxSoC is M among all affected labels and thus obviously irrelevant among them. Let B be the set of labels that have already been affected by overcharging. Because maxSoC cannot exceed M, no label has a higher maxSoC value than the ones in B. This remains true in the further course of P, regardless of which operations are performed: Traversing an edge without overcharging shifts maxSoC by the same value for all labels. A charging station will set maxSoC to M for all labels, including the ones in B, so it will still remain irrelevant among them. If overcharging occurs again, it affects all labels whose maxSoC value would otherwise exceed M after traversing the edge. Because the labels in B have the highest maxSoC value among the label set, they will be affected again now. Further labels may be added to B, but since maxSoC is set to M among both the old labels and the new labels in B, it will still be irrelevant among all of B.
4. As observed earlier, a charging station sets maxSoC $= M$ for all labels, regardless of the operations on the previous path. Thus, all labels are certainly maxSoC-equal directly after encountering a charging station. This remains true in the further course

of the path: Edges without overcharging in the remaining path will only shift maxSoC by the same value for all labels, leaving them maxSoC-equal. Overcharging occurs when maxSoC would otherwise exceed M after traversing an edge. Since maxSoC is already equal for all labels, overcharging can only affect all labels at once, so the labels remain maxSoC-equal in this case as well.

□

With these observations, we can partition our label sets. Consider a label set L' at a vertex s, a path $P = (s = v_1, v_2, \ldots, v_k = t)$ and the label set $L := \text{trim}(L' \circ P)$ at t. We can divide L into the following disjoint groups:

L_{0x}: Labels that have encountered partial undercharging but no maxSoC-omitting operation in P. cons + minSoC is irrelevant among this group.

L_{0M}: Labels that have encountered both partial undercharging and a maxSoC-omitting operation in P, but not complete overcharging. Since cons + minSoC and maxSoC are both irrelevant among this group, it consists of at most one label.

L_{xx}: Labels that have encountered neither a cons + minSoC-omitting operation nor a maxSoC-omitting operation in P. This is the only group in which no criterion is irrelevant.

L_{xM}: Labels that have encountered a maxSoC-omitting operation but no cons + minSoC-omitting operation in P. maxSoC is irrelevant among this group.

L_{MM}: Labels that have encountered complete overcharging in P. Since cons + minSoC and maxSoC are both irrelevant among this group, it consists of at most one label.

The indices of each group denote the effect that the operations in P have had on the labels. The first index denotes the effect on minSoC: 0 means that partial undercharging has occured, which sets minSoC = 0, M means that complete overcharging has occured, which sets minSoC = M, and x means that neither has occured and that minSoC may have any value (including M). The second index denotes the effect on maxSoC: M means that overcharging has occured, which sets maxSoC = M, whereas x means that it has not occured and that maxSoC may have any value. A group exists for any combination of indices except L_{Mx}, which is impossible since it would have required minSoC to be higher than maxSoC at some point in P.

The groups are disjoint by definition, but some of the groups may be empty. For example, if P contains neither under-/overcharging for any label nor charging stations, L_{xx} will contain all labels and the other groups will be empty. If the path contains a charging station, maxSoC will be omitted per Lemma 5.4, so L_{0x} and L_{xx} will become empty.

5.3 Chain Graphs

In the following we will introduce a class of graphs called *chain graphs* and, using the observations made in the previous section, prove that the size of the label sets in these graphs cannot exceed $\mathcal{O}(m^2)$, where m is the number of edges in the graph. To begin, we give a definition of chain graphs:

Definition 5.5.

- *A path graph is a graph that consists entirely of the vertices and edges of a path* $P = (v_1, v_2, \ldots, \ldots v_k)$.

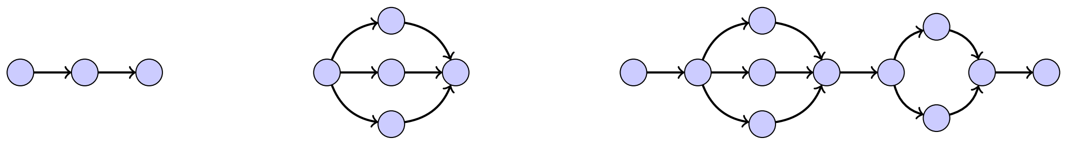

Figure 5.2: Examples of a path graph (left), a chain segment of width 3 (center) and a chain graph (right).

- *A chain segment C is the union of a set S of (pairwise) disjoint path graphs, where the first vertices of each path graph are merged together into a single source vertex s and the last vertices of each path graph into a single sink vertex t. We call $w(C) := |S|$ the width of the chain segment.*
- *A chain graph G is the union of a sequence $S = (C_1, C_2, \dots, C_k)$ of (pairwise) disjoint chain segments, where the sink of each C_i $(1 \leq i < k)$ is merged with the source of C_{i+1}. We call the source of C_1 the source of G and the sink of C_k the sink of G.*

In Section 5.1, we used an example graph (see Figure 5.1) to demonstrate that the label sets may have exponential size for general multi-objective shortest path search problems. This example graph fulfills the definition of a chain graph where every chain segment has width 2. Chain graphs have the same general structure as this example graph, but instead of single edges between the individual vertices of the sequence, there may now be entire paths. Additionally, the number of paths is not restricted to two and may be arbitrarily large. Several examples of chain graphs are shown in Figure 5.2.

To find an upper bound for the size of the label sets in the graph, we can iterate through the chain segments: If we know the structure of the label set at the source of a chain segment, we can use the observations from the previous section to characterize the label sets at the end of each path in the segment. When merged together, these label sets become the label set of the segment's sink, which is the source of the next segment.

When the label sets of each path are merged together at the end of a segment, labels from different paths may dominate each other. We observe that this is always the case for labels that encountered no charging station or over-/undercharging in the segment:

Lemma 5.6. *Let C be a chain segment with a source s, a sink t and two paths $P = (s = u_1, u_2, \dots, u_k = t)$ and $Q = (s = v_1, v_2, \dots, v_k = t)$. Let L_s be the label set of s, L_t the label set of t, and $L_P := L \circ P$ and $L_Q := L \circ Q$ the label sets of u_k and v_k, respectively, before being merged into L_t and before applying dominance. Let ω_P and ω_Q be the sum of all edge weights in P and Q, respectively, and without loss of generality, let $\omega_P \leq \omega_Q$.*

For every label $\ell \in L_s$ that encounters over-/undercharging or a charging station neither in P nor in Q, the following is true: $(\ell \circ P) \propto (\ell \circ Q)$.

Proof. Because ℓ does not encounter undercharging, we know that $\ell \circ P \in L_P$ and $\ell \circ Q \in L_Q$. With $\omega_P \leq \omega_Q$ we observe:

- $(\ell \circ P).\operatorname{cons} = \ell.\operatorname{cons} + \omega_P \leq \ell.\operatorname{cons} + \omega_Q = (\ell \circ Q).\operatorname{cons}$.
- $(\ell \circ P).\operatorname{minSoC} = \ell.\operatorname{minSoC} - \omega_P \geq \ell.\operatorname{minSoC} - \omega_Q = (\ell \circ Q).\operatorname{minSoC}$.
- $(\ell \circ P).\operatorname{cons} + (\ell \circ P).\operatorname{minSoC} = \ell.\operatorname{cons} + \omega_P + \ell.\operatorname{minSoC} - \omega_P =$ $\ell.\operatorname{cons} + \ell.\operatorname{minSoC} = \ell.\operatorname{cons} + \omega_Q + \ell.\operatorname{minSoC} - \omega_Q =$ $(\ell \circ Q).\operatorname{cons} + (\ell \circ Q).\operatorname{minSoC}$.
- $(\ell \circ P).\operatorname{maxSoC} = \ell.\operatorname{maxSoC} - \omega_P \geq \ell.\operatorname{maxSoC} - \omega_Q = (\ell \circ Q).\operatorname{maxSoC}$.

In total, this means that $(\ell \circ P) \propto (\ell \circ Q)$ and thus $\ell \circ Q \notin L_t$. □

We can generalize this observation for two paths to apply to a whole chain segment. This gives us an upper bound for the number of labels that encountered neither over-/undercharging nor a charging station:

Lemma 5.7. *Let G be a chain segment with a source s and a sink t. Let L_s be the label set of s, L_t the label set of t, and $L_{xx} \subseteq L_t$ the set of all labels in L_t that have encountered neither a charging station nor over-/undercharging since L_s. Then, the following holds true: $|L_{xx}| \leq |L_s|$.*

Proof. Assume that $|L_{xx}| > |L_s|$. That means that there must be two labels $\ell_1, \ell_2 \in L_{xx}$ that originate from a common label in L_s but took different paths. Per Lemma 5.6, one of these labels must dominate the other, so L_{xx} cannot contain both. □

When labels that did encounter a charging station or over-/undercharging are involved, however, it is possible that multiple labels that originate from the same label at the source but took different paths all remain in the label set of the sink without dominating each other. For example, consider a path that contains a charging station but has a higher total energy consumption than another path without a charging station. The labels from the former path will be superior according to maxSoC, while the labels from the latter will be superior according to cons, so the labels will not dominate each other.

To make the deduction of an upper limit for the size of the label sets simpler, we will assume in the following that two labels from different paths of a chain segment only dominate each other if neither encountered a charging station nor over-/undercharging in the segment. While there are situations where labels without this restriction dominate each other, we can ignore those cases because doing so will only increase our upper bound for the size of the label sets.

The observations from the previous chapter allowed us to divide the label sets at the end of each path in a chain segment into groups according to which dominance criteria are irrelevant among them. In the following Lemma, we will show that there is another way to group the label set at the sink of a chain segment so that dominance criteria become irrelevant among some groups:

Lemma 5.8. *Let C be a chain segment with a source s, a sink t and w s-t-paths $P_1, \dots, P_w$. Let L_s be the label set at s and L^i the label set at the end of each path P_i ($1 \leq i \leq w$). Let $U := \bigcup_{i=1}^{p} L^i_{0x} \cup L^i_{xx}$ be the union of all label groups among which* maxSoC *was not omitted and $V := \bigcup_{i=1}^{p} L^i_{xx} \cup L^i_{xM}$ the union of all label groups among which* $\text{cons} + \text{minSoC}$ *was not omitted. For every label $\ell \in L_s$, let $L_\ell := \{\ell \circ P_i \mid i \in \{1, \dots, p\}\}$ be the set of labels at the end of C that originate from ℓ. Then, the following is true:*

1. maxSoC *and* cons *are equivalent among $U_\ell := U \cap L_\ell$.*
2. *$V_\ell := V \cap L_\ell$ is (*$\text{cons} + \text{minSoC}$*)-equal.*

Proof.

1. U_ℓ contains only labels from groups among which maxSoC has not been omitted. This means that the labels in U_ℓ have encountered neither overcharging nor a charging station. Therefore, if ω_i is the sum of all edge weights in P_i, we know that $(\ell \circ P_i).\text{cons} = \ell.\text{cons} + \omega_i$ and $(\ell \circ P_i).\text{maxSoC} = \ell.\text{maxSoC} - \omega_i$. Together, this yields for every label $\ell' \in U_\ell$:

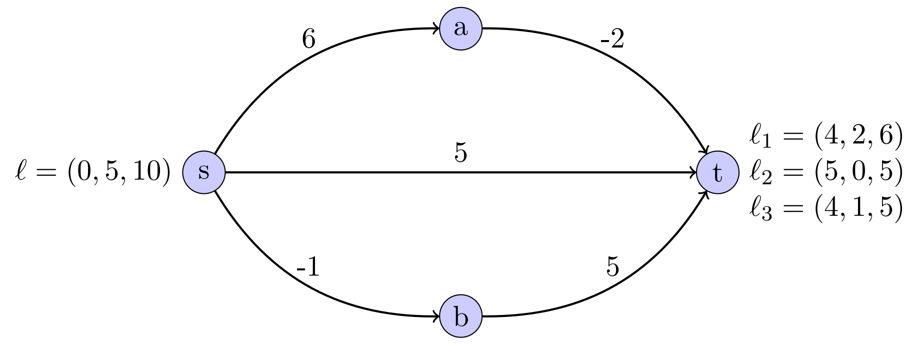

Figure 5.3: An illustration of case 1 of Lemma 5.8. This chain segment consists of three paths, denoted from the top to the bottom as P_1, P_2 and P_3. Picking the label $\ell \in L_s$, $L_\ell \subseteq L_t$ consists of the labels ℓ_1, ℓ_2 and ℓ_3, all of which originate from ℓ. ℓ_1 encountered partial undercharging but no maxSoC-omitting operation and is therefore part of L^1_{0x}. ℓ_2 encountered no criterion-omitting operation at all and is thus part of L^2_{xx}. ℓ_3 encountered no undercharging but partial overcharging and is therefore part of L^3_{xM}. U_ℓ consists of the labels in L_ℓ that encountered no maxSoC-omitting operation, which are ℓ_1 and ℓ_2. In compliance with the statement of the Lemma, we observe: $\ell_1.\operatorname{cons} + \ell_1.\operatorname{maxSoC} = \ell_2.\operatorname{cons} + \ell_2.\operatorname{maxSoC} = \ell.\operatorname{cons} + \ell.\operatorname{maxSoC} = 10$, but $\ell_3.\operatorname{cons} + \ell_3.\operatorname{maxSoC} = 9$. maxSoC and cons are equivalent among U_ℓ, but not among all of L_ℓ.

$$\ell'.\operatorname{cons} + \ell'.\operatorname{maxSoC} = (\ell.\operatorname{cons} + \omega_i) + (\ell.\operatorname{maxSoC} - \omega_i) = \ell.\operatorname{cons} + \ell.\operatorname{maxSoC} =: c.$$

The value c is constant among U_ℓ and thus maxSoC and cons are equivalent according to Definition 5.1. An illustration of this case can be seen in Figure 5.3.

2. V_ℓ contains only labels from groups among which $\operatorname{cons} + \operatorname{minSoC}$ has not been omitted. This means that the labels in V_ℓ have not encountered undercharging. Since undercharging is the only operation that changes the value of $\operatorname{cons} + \operatorname{minSoC}$, we know that: $\forall \ell' \in V_\ell : \ell'.\operatorname{cons} + \ell'.\operatorname{minSoC} = \ell.\operatorname{cons} + \ell.\operatorname{minSoC}$. This value is constant among V_ℓ and thus V_ℓ is $(\operatorname{cons} + \operatorname{minSoC})$-equal.

□

By combining these two approaches, we can examine the label sets on each path of a new chain segment and find groups in which two dominance criteria become irrelevant. Because only one label may survive from each such group, this gives us a restriction on the number of labels that each chain segment may add.

Lemma 5.9. *Let $G = (V, E)$ be a chain graph consisting of a sequence of chain segments $T = (C_1, C_2, \dots, C_k)$. For each segment C_i, let w_i be the width of C_i and $T_i = (P_{i,1}, P_{i,2}, \dots, P_{i,w_i})$ an enumeration of the paths in C_i. Let $L^{i,j}$ be the label set at the end of each path $P_{i,j}$ and L^i the label set at the sink of C_i, which is the result of merging the label sets of all paths and applying dominance. Let $L^0 := \{(0, S, S)\}$ be the Pareto set at the source of C_1, which consists of only one label.*

For every label set $L^{i,j}$ at the end of a path $P_{i,j}$ in segment C_i, the following is true:

- $|L^{i,j}_{0x}| \leq 1 + 3 \sum\limits_{k=1}^{i-1} w_k$.

- $|L^{i,j}_{0M}| \leq 1$.
- $|L^{i,j}_{xx}| \leq |L_{i-1}|$.
- $|L^{i,j}_{xM}| \leq 1 + 3\sum\limits_{k=1}^{i-1} w_k$.
- $|L^{i,j}_{MM}| \leq 1$.

After merging these label sets into L_i and applying dominance, the following is true:

$$|L_i| \leq |L_{i-1}| + 2w_i(2 + 3\sum_{k=1}^{i-1} w_k).$$

Proof. First we prove that the upper bound for $|L^i|$ holds true if the upper bounds given for the groups in $L^{i,j}$ hold true: We know that $L^{i,j}$ is the union of the five groups $L^{i,j}_{0x}$, $L^{i,j}_{0M}$, $L^{i,j}_{xx}$, $L^{i,j}_{xM}$ and $L^{i,j}_{MM}$. This gives us:

$$|L^{i,j}| \leq |L^{i-1}| + 2(2 + 3\sum_{k=1}^{i-1} w_k).$$

There are w_i paths in C_i, so if none of these labels dominated each other, our upper bound for $|L^i|$ would be:

$$|L^i| \leq w_i(|L^{i-1}| + 2(2 + 3\sum_{k=1}^{i-1} w_k)).$$

We divided $L^{i,j}$ such that $L^{i,j}_{xx}$ contains labels that have encountered neither a (cons + minSoC)-omitting operation nor a maxSoC-omitting operation in C_i. According to Lemma 5.7, L^i can contain at most $|L^{i-1}|$ labels from all groups $L^{i,j}_{xx}$ $(1 \leq j \leq w_i)$. The labels in the other groups have all encountered a (cons + minSoC)-omitting operation or a maxSoC-omitting operation, so it is possible that none of these labels are dominated in L^i. In total this gives us:

$$|L^i| \leq |L^{i-1}| + 2w_i(2 + 3\sum_{k=1}^{i-1} w_k).$$

Now we can prove the rest of the lemma by induction over the segments $C_1, \ldots, C_k$ in T:

Base case ($i = 1$):

Because $|L^0| = 1$, the label set $L^{1,j}$ at the end of each path $P_{1,j}$ in C_1 can have at most one label. This implies the statement of the lemma, since $\sum_{k=1}^{0} w_k = 0$ holds.

Inductive step ($i \to i + 1$):

Assume that the upper bounds given by the statement hold true for C_i. For each path $P_{i+1,j}$ in C_{i+1} we analyze the label set $L^{i+1,j}$ at the end of the path:

- $L^{i+1,j}_{xx}$ encountered no criterion-omitting operations, so all labels from L^i are retained. Therefore, $|L^{i+1,j}_{xx}| \leq |L^i|$ holds.
- Both cons + minSoC and maxSoC are irrelevant in $L^{i+1,j}_{0M}$ and $L^{i+1,j}_{MM}$, so each of these groups contains at most one label. Therefore, $|L^{i+1,j}_{0M}| \leq 1$ and $|L^{i+1,j}_{MM}| \leq 1$ hold.

- Among $L_{0x}^{i+1,j}$, cons + minSoC becomes irrelevant.
  - Let $U \coloneqq \bigcup_{j=1}^{w_i} L_{0x}^{i,j} \cup L_{xx}^{i,j}$ be the union of all groups at the end of C_i among which maxSoC was not omitted. For every label $\ell \in L^{i-1}$, let $L_\ell \coloneqq \{\ell \circ P_{i,j} \mid j \in \{1, \dots, w_i\}\}$ be the set of labels at the end of C_i that originate from ℓ. To the union $U_\ell \coloneqq U \cap L_\ell$ we can apply Lemma 5.8, which tells us that maxSoC is irrelevant among each U_ℓ. After traversing $P_{i+1,j}$, consider $(U_\ell \circ P_{i+1,j}) \cap L_{0x}^{i+1,j}$, which contains all labels from U_ℓ that have become part of $L_{0x}^{i+1,j}$. Obviously maxSoC is still irrelevant among these labels, and because cons + minSoC has now become irrelevant among them as well, at most one label remains in each such group. Since we constructed one such group for every label in L^{i-1}, the remaining labels originate from a subset of L^{i-1}. Per induction hypothesis, each $L_{0x}^{i,j}$ group in C_i contained at most $1 + 3\sum_{k=1}^{i-1} w_k$ labels, all of which originated from L^{i-1}. Because the remaining labels in $L_{0x}^{i+1,j}$ group all originate from a subset of L^{i-1}, we can apply the same upper bound.
  - In each $L_{xM}^{i,j}$ group in C_i, maxSoC was already irrelevant. Since cons + minSoC is now irrelevant as well, only one label may remain at most in $L_{0x}^{i+1,j}$ from each of those groups. Since there were w_i paths in C_i, these are at most w_i labels.
  - The labels from the $L_{0M}^{i,j}$ and $L_{MM}^{i,j}$ groups in C_i may all be retained. Since there were w_i paths in C_i, these are at most $2w_i$ labels.
  - In total this yields: $|L_{0x}^{i+1,j}| \leq 1 + 3\sum\limits_{k=1}^{i-1} w_k + 3w_i = 1 + 3\sum\limits_{k=1}^{i} w_k$.
- Among $L_{xM}^{i+1,j}$, maxSoC becomes irrelevant. The argument for this case is equivalent to the $L_{0x}^{i+1,j}$ case, with the roles of $L_{xM}^{i+1,j}$ and $L_{0x}^{i+1,j}$, $L_{0x}^{i,j}$ and $L_{xM}^{i,j}$, as well as cons + minSoC and maxSoC, swapped. This yields: $|L_{xM}^{i,j}| \leq 1 + 3\sum\limits_{k=1}^{i} w_k$.

We see that the upper bounds also hold true for C_{i+1}. □

Using this upper bound for the number of labels added per segment, we can easily deduct an upper bound on the total number of labels:

Theorem 5.10. *Let $G = (V, E)$ be a chain graph consisting of a sequence of chain segments $T = (C_1, C_2, \dots, C_p)$. For each segment C_i, let w_i be the width of C_i and L_i the label set at the sink of C_i. Let $|L|$ the maximum size of any label set in V. Then we have $|L| \in \mathcal{O}(m^2)$, where $m = |E|$.*

Proof. Per Lemma 5.9, we know that the size of the label set after i chain segments is:

$$|L_i| \leq |L_{i-1}| + 2w_i(2 + 3\sum_{k=1}^{i-1} w_k).$$

Because every path in a chain segment of non-zero width must contain at least one edge, we know that:

$$\sum_{k=1}^{p} w_k \leq m.$$

With this, we can find an upper bound for L_i:

$$|L_i| \leq |L_{i-1}| + 2w_i(2 + 3\sum_{k=1}^{i-1} w_k) \leq |L_{i-1}| + 2w_i(2 + 3m).$$

This yields for every segment C_i:

$$|L_i| \leq \sum_{k=1}^{i} 2w_k(2 + 3m) \leq 2(2 + 3m)\sum_{k=1}^{p} w_k \leq 2(2 + 3m)m \in \mathcal{O}(m^2).$$

□

The worst-case complexity we deducted for our algorithm in Chapter 4.4 was $\mathcal{O}(n|L|\log n + m|L|^2)$ for a Fibonacci heap and $\mathcal{O}((n + m)|L|\log n + m|L|^2)$ for a binary heap, where $|L|$ was the maximum size of any label set in the graph G. If G is a chain graph and only contains regular charging stations, Theorem 5.10 tells us that $|L| \in \mathcal{O}(m^2)$ holds. With this, the worst-case complexity of our algorithm becomes $\mathcal{O}(m^2 n \log n + m^5)$ for a Fibonacci heap and $\mathcal{O}(m^2(n + m)\log n + m^5)$ for a binary heap.

In a road network, the edges represent road segments and the vertices represent end points or intersections of road segments. This means that every vertex is connected to at least one edge and thus $n \in \mathcal{O}(m)$ holds. Using this relation to substitute n, the worst-case complexity becomes $\mathcal{O}(m^5)$ for both heap types.

6. Profile Shortcut Algorithm

The algorithm we developed in Chapter 4 solves the EEVRC problem, but it requires the introduction of Pareto sets in order to handle the addition of charging stations. As observed in Chapter 5, this dramatically increases the worst-case complexity of the algorithm. In this chapter, we will develop an alternative algorithm that avoids the use of Pareto sets by precalculating the shortest feasible paths between all pairs of charging stations. This information can be used to significantly reduce the algorithm's search space, leading to faster running times in practice.

6.1 Basic Approach for Battery Swapping Stations

In [SF12], a version of this algorithm was introduced that only considers battery swapping stations as a possible method of recharging. The basic idea is to precalculate the shortest feasible paths between all pairs of charging stations. In order to answer an s-t-query, we then only need to calculate the path from s to the first charging station and from the last charging station to t; the remainder of the path can be found by using the precalculated information.

Given a graph $G = (V, E)$ with a set of charging stations $\mathrm{CS} \subseteq V$, we precalculate an auxiliary graph $H = (\mathrm{CS}, E_H)$ consists only of the charging stations. For every pair $u, v \in \mathrm{CS}$ of charging stations, H contains an edge $e = (u, v)$ if and only if v can be reached from u without recharging, provided that the vehicle starts at u with a full battery, i.e., $b(u) = M$. For the edge weight $\mathrm{cons}(e)$, we choose the amount of energy that is consumed on the shortest feasible u-v-path. We call this edge a *shortcut* between u and v, since it compresses the shortest feasible u-v-path into a single edge. Accordingly, we call H the *shortcut graph* of G. We can calculate the outgoing shortcut edges from each charging station $u \in \mathrm{CS}$ by performing a Dijkstra query from u that takes battery constraints into account. Once the query has finished, we add shortcut edges according to the distances that it calculated to the other charging stations: For every other charging station $v \in \mathrm{CS}$ with $\mathsf{dist}[v] < \infty$, we add an edge $e = (u, v)$ to E_H with the weight $\mathrm{cons}(e) = \mathsf{dist}[v]$. The shortcut graph H can thus be calculated by performing one Dijkstra query for each charging station in G.

Once we have calculated the shortcut graph H (which only needs to be done once for every graph), we can answer an s-t-query as follows:

- First we perform a single-source Dijkstra query from s that takes battery constraints into account but does not allow for recharging. If this query reaches t, then no charging stations are required and we can simply return the found path. Otherwise, we must continue our query on the shortcut graph. We save the set $\mathrm{CS}_s \subseteq \mathrm{CS}$ of charging stations that were reached from s, along with their distances from s as given by the array $\mathsf{dist}[\cdot]$.
- We can use H to calculate the path up to the last charging station before reaching t, but unless t is a charging station itself, it does not contain any information about the path from the last charging station to t. Therefore, we add t to H and calculate a shortcut edge $e = (v, t)$ for every charging station $v \in \mathrm{CS}$ from which can be reached without recharging. This can be done by performing a single-target backwards Dijkstra query from t. For every charging station $v \in \mathrm{CS}$ with $\mathsf{dist}[v] < \infty$, we add $e = (v, t)$ to E_H with the weight $\mathrm{cons}(e) = \mathsf{dist}[v]$.
- Now H can be used to calculate the remainder of the path, starting at CS_s. We know that every time a battery swapping station $u \in \mathrm{CS}$ is used, the SoC of the vehicle is reset to M. This is exactly the SoC that the outgoing shortcut edges from u require, so we can use them to continue the query. We thus find the shortest feasible s-t-path as follows: We reinsert each charging station $v \in \mathrm{CS}_s$ into the priority queue, using $\mathsf{dist}[v]$ as the key. Then we continue the Dijkstra query, but now using the edges in H instead of the ones in G. At each vertex, we reset the SoC to M in order to use the outgoing shortcut edges. Once the query has finished, $\mathsf{dist}[t]$ is the length of the shortest feasible s-t-path.

We see that this algorithm does not require Pareto sets for deciding whether to use a battery swapping station or not. In fact, due to the structure of our shortcut graph H, this decision is no longer necessary because superfluous charging stations are automatically skipped: Consider a shortest feasible s-t-path that visits three consecutive charging stations $u, v, w \in \mathrm{CS}$. Assume that it is necessary to charge at u and w, but that w can be reached directly from u without recharging, so charging at v is unnecessary. Since w can be reached directly from u, the shortcut graph H must contain an edge (u, w) in addition to the edges (u, v) and (v, w). Both the single edge (u, w) and the pair of edges (u, v) and (v, w) represent the same path in G, but (u, w) skips over v without recharging there. If charging at v causes overcharging, $\mathrm{cons}((u, w)) < \mathrm{cons}((u, v)) + \mathrm{cons}((v, w))$ holds and (u, w) will lead to a lower energy consumption at w than (u, v) and (v, w). Otherwise, $\mathrm{cons}((u, w)) = \mathrm{cons}((u, v)) + \mathrm{cons}((v, w))$ holds. Because the vehicle reaches u before v, the edge (u, w) is relaxed before (v, w). Because (v, w) does not improve the energy consumption at w, it is discarded and the query finds the path that skips recharging at v.

Because Pareto sets are not required, the algorithm can simply use regular Dijkstra queries for electric vehicles in all steps. Precalculating H requires one Dijkstra query for every charging station in G. Each s-t-query requires three Dijkstra queries: One from s in G, one to add t to H, and one to reach t in H. Thus, the entire algorithm has polynomial complexity in the size of graph when applying potential shifting. Additionally, its search space is fairly small: The queries that are performed in G do not allow for recharging, so they will abort once the battery is depleted, regardless of the distance between s and t. Only the query in H allows for recharging, but for realistic road networks, H contains much fewer vertices than G.

Path Retrieval. Each shortcut edge $e = (u, v)$ represents a shortest u-v-path $P_e = (u, \ldots, v)$, but H no longer stores any information about the other vertices in P_e besides u and v. However, we need to know these vertices if we want our algorithm to output the shortest s-t-path. Thus, when adding e to H, we retrieve P_e by following the parent pointer of v and store it alongside e. During our Dijkstra query in H, each shortcut edge $e = (u, v)$ now represents an entire path, so we must adapt our parent pointers accordingly: Instead of pointing only to u, we save the entire path P_e between u and v. We can then retrieve the s-t-path by following the parent pointers and concatenating the resulting paths.

6.2 Incorporating Other Charging Station Types

So far, the algorithm makes the assumption that every charging station always resets the SoC of the vehicle to exactly M. This is true for battery swapping stations, but not for other types of charging stations, where we can choose how much energy to charge. While regular charging stations still allow us to charge to M, this might cause overcharging in the further course of the path that must be avoided. Furthermore, superchargers can at most charge to $0.8 \cdot M$. This means that we can no longer know beforehand what the resulting SoC at a given charging station will be. However, we need this information to calculate the weights of our shortcut edges.

In order to solve this problem, we use profile queries as introduced in Chapter 2.4.3. Whereas a regular Dijkstra query calculates the shortest s-t-path for a given SoC $b(s)$ at s, a profile query calculates the shortest s-t-path for every possible SoC at s. We thus replace the static edge weights of our shortcut edges with energy consumption profiles that we calculate via profile queries: During the preprocessing phase, we perform a profile query from every charging station $u \in \text{CS}$. For every other charging station $v \in \text{CS}$ that is reachable from u for any SoC in $B = [0, M]$, we add a shortcut edge $e = (u, v)$ to E_H, using the calculated u-v-profile as the edge weight. This profile gives us the shortest u-v-path for any SoC at u. Likewise, we perform a backwards profile query from t to calculate the profiles for the shortcut edges from each charging station to t.

When performing the Dijkstra query in the shortcut graph H, we now need to decide at every charging station how much energy to charge there. As observed in Chapter 4, this decision can only be made once we have reached the next charging station that is used. In order to solve this problem, our previous algorithm maintained an entire interval of possible values for the amount of charged energy and only calculated the actual amount to charge once the query had finished and the shortest feasible path was known. Due to the structure of our shortcut graph H, this is no longer necessary: As observed in the previous section, the query will automatically skip charging stations if it is not necessary to charge at them. Therefore, we can assume that all charging stations that are visited during our query are also used.

When relaxing an edge $e = (u, v)$, we check if e is traversable without recharging at u. If this is the case, we do not need to relax e, since it is not part of the optimal solution. Otherwise, we must decide how much energy to charge at u. As observed in Chapter 4, the optimal strategy is to avoid overcharging in the further course of the path, but otherwise as much as possible. Because we know that the vehicle will recharge again at v, we only need to avoid overcharging on e. The optimal SoC at u can thus be found by evaluating the profile p of u: If $b_{\min} \in B$ is the SoC with which the vehicle reaches u and $b_{\max} \in B$ is the maximal SoC that the charging station can charge to, then the optimal SoC is the highest SoC $b \in [b_{\min}, b_{\max}]$ that minimizes the energy consumption $p(b)$ on e. After finding this optimal SoC b, we can then traverse e with the energy consumption $p(b)$.

Because our algorithm uses a shortcut graph whose edge weights are energy consumption profiles, we call it the Profile Shortcut Algorithm (PSA). Due to the involvement of profile queries, it is label-correcting even after applying potential shifting and we can no longer guarantee a polynomial running time in the size of G.

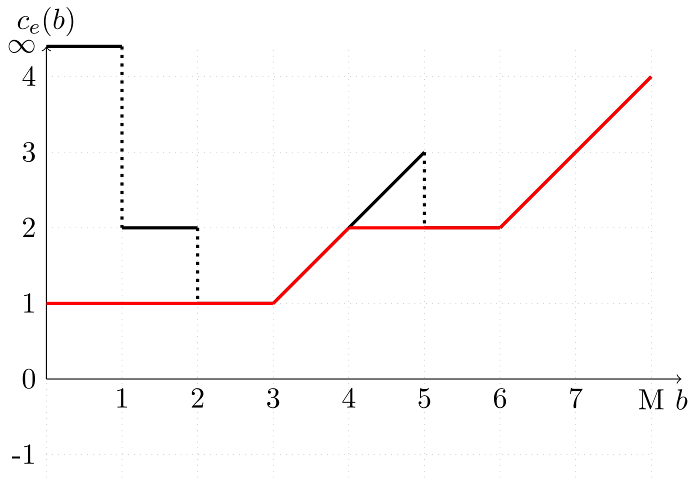

Figure 6.1: An example of an energy consumption profile (black) and its trimmed profile (red). Note that the trimmed profile is monotonically increasing.

Path Retrieval. Since the shortcut edges now represent entire energy consumption profiles, there is no longer one definite shortest path associated with every shortcut edge $e = (u, v)$. Instead, one shortest path is associated with every supporting point of e. Therefore, when adding e to H, we need to retrieve the shortest path for every supporting point in the profile of e and store it alongside e. During our Dijkstra query in H, the path associated with a shortcut edge $e = (u, v)$ is now the one associated with the supporting point whose interval includes the current SoC at u.

Trimming Profiles. When deciding how much to charge at a given charging station $u \in \text{CS}$ before traversing an edge $e = (u, v)$, we consider the interval $[b_{\text{min}}, b_{\text{max}}]$, where b_{min} is the SoC with which the vehicle reaches u and b_{max} is the maximum SoC that the charging station at u can charge to. In this interval, we choose the highest SoC that minimizes the energy consumption on e. We can use this knowledge to simplify our energy consumption profiles: Given a profile p, we know for every SoC $b_{\text{min}} \in B$ that the best possible energy consumption is given by

$$\text{cons}_{\text{min}}(b_{\text{min}}) := \min_{b \in [b_{\text{min}}, b_{\text{max}}]} (p(b)).$$

We know that the energy consumption on e after recharging is $\text{cons}_{\text{min}}(b_{\text{min}})$, so we can define a new profile function p' that uses this value instead of $p(b)$:

$$p'(b) := \begin{cases} \text{cons}_{\text{min}}(b) & \text{if } b \leq b_{\text{max}} \\ p(b) & \text{otherwise.} \end{cases}$$

As shown in Figure 6.1, p' is monotonically increasing in $[0, b_{\text{max}}]$. If we use p' instead of p as the profile for e, the energy consumption on e after recharging at u is directly given by $p'(b_{\text{min}})$. However, we also need to know how much to energy charge to achieve this energy consumption. Therefore, for every supporting point $(b_{\text{min}}, p'(b_{\text{min}}))$ in the new profile p' we save the SoC that the vehicle must charge to at u to achieve the energy consumption $p'(b_{\text{min}})$. This SoC can be calculated as

$$\text{SoC}_{\text{max}}(b_{\text{min}}) := \max(\{b \in [b_{\text{min}}, b_{\text{max}}] \mid p'(b) = p'(b_{\text{min}})\}).$$

7. Evaluation

In this chapter we evaluate our implementations of the algorithms introduced in Chapter 4 and 6 on real road network data. First we discuss the experimental setup and the input data used for our experiments. Then we evaluate the performance of the two algorithms and their various speedup techniques.

7.1 Experimental Setup and Input Data

We evaluate our algorithms on road network data kindly provided to us by the PTV AG for scientific use. This data contains information about road segments and intersection, including geographical coordinates and road types. For each road type, an average driving speed is given. By multiplying this value with the length of the edges, we obtain an average driving time for each road segment. We extracted the road networks of Luxembourg and Germany from this data, obtaining the two graphs *luxembourg* and *germany*.

In order to compute energy-optimal routes, we need an edge weight function that maps the energy consumption to each road segment. Among other factors, the energy consumption of an electric vehicle depends of the slope of the road segment. We compute the slopes for the edges in our graphs by using the elevation data provided by the Shuttle Radar Topography Mission (SRTM), which is freely available from the CGIAR Consortium for Spatial Information [1]. This data covers over 80% of the globe with a precision of 90 meters. Parts of the graphs which are not covered by this data are removed.

We compute the energy consumption data using PHEM (Passenger car and Heavy duty vehicle Emission Model) [HRZL09], which is developed by the Graz University of Technology. This model computes the energy consumption for a variety of electric vehicle types and driving situations, taking into account the road category, driving speed and slope of a road segment. We mapped the road types provided by the PTV AG data to the PHEM road

[1]http://srtm.csi.cgiar.org/

Table 7.1: An overview of the graphs used for our experiments.

Graph	# vertices	# edges	# edges with cons ≤ 0	# charging stations
luxembourg	36 457	81 950	12 947 (15.79%)	37 (0.10%)
germany	4 692 091	10 805 429	1 119 710 (10.36%)	1 970 (0.04%)

categories using a heuristic previously used by Baum et al. [BDPW13]. As a vehicle type we choose the Peugeot iOn model, which is equipped with a 16kWh battery. We use this battery capacity for our experiments on the *germany* graph. On the *luxembourg* graph, a battery capacity of 16kWh is sufficient for traversing the entire graph without needing to recharge the battery. In order to study the effect of charging stops, we therefore use a battery capacity of 4kWh on this graph.

In addition to the road networks and energy consumption data, we also need information about the location of charging stations. We obtain this information from ChargeMap[2], which lists the locations of many charging stations worldwide. We map the location of each charging station to the vertex which is closest in our graph, discarded it if the distance to the closest vertex exceeds 20 meters. Using this method, we obtain coordinates for 1 970 charging stations in the *germany* graph and 37 charging stations in the *luxembourg* graph. To each of these charging stations we randomly assign a charging station type. Unless otherwise noted, we use a distribution of 80% regular charging stations, 10% superchargers and 100% battery swapping stations.

An overview of the data used in our experiments is given in Table 7.1, including information about the size of the graphs, the number of edges with a negative energy consumption and the number of charging stations.

7.2 Experiments

We now evaluate the performance of our algorithms. All algorithms were implemented in C++ and compiled using g++ version 4.8.3 (64 bit) with optimization level 3. The experiments were conducted on machine with two 8-core Intel Xeon E5-2670 processors clocked at 2.6GHz and 64GB of DDR3-1600 RAM. In order to ensure reproducibility of the experiments, only one algorithm was running at any given time. Unless otherwise noted, each algorithm used only a single core.

7.2.1 Charging Label Algorithm

We start by evaluating the Charging Label Algorithm introduced in Chapter 4. For the purpose of comparison, we also evaluate an implementation of CLA that uses the distance and SoC functions introduced in Chapter 4.1. We call this version of the algorithm Naive CLA. In addition to the basic algorithm, we also developed three speedup techniques: By calculating the maximum recuperation possible at each vertex, we were able to prune the search by discarding labels whose energy consumption is too high to improve the current consumption at the target. The same information was also used to develop stricter dominance criteria. Finally, we used potential shifting to introduce a stopping criterion.

In order to test the performance of CLA as well as its speedup techniques, we evaluated the same randomly chosen 100 queries on the *germany* graph for Naive CLA, basic CLA without speedups, as well as every useful combination of speedups. The results are shown in Tables 7.2 and 7.3. Note that combining the pruning technique and potential shifting is not useful: The pruning technique only starts discarding labels once the target vertex has been visited for the first time. After applying potential shifting, however, the entire query can already be aborted once the target vertex is reached, so the pruning technique will never take effect. Therefore, this combination was not evaluated.

We observe that CLA without any speedups is almost three times as fast as Naive CLA, improving the average running time from 63 seconds to 22 seconds. The reason for this vast improvement in performance is that the operations for traversing an edge and using

[2]http://chargemap.com/

Table 7.2: This table shows the performance of CLA with various combinations of speedup techniques. For this experiment, the same 100 random queries on the *germany* graph were evaluated for each configuration of the algorithm. This table shows the results for three out of seven configurations that were tested; the remaining results are shown in Table 7.3.

Algorithm	Naive CLA	CLA	
Potentials?	✗	✗	✗
Pruning?	✗	✗	✗
Improved dominance?	✗	✗	✓
Time [s]	63.378	22.170	19.006
Avg. label set size	3.07	3.57	2.98
Max. label set size	37	41	16
Visited vertices	4 692 091	4 692 091	4 692 091
Visited labels	19 668 609	22 773 258	19 577 307
Visited labels per vertex	4.19	4.85	4.17
Rescanned labels	556 399	603 390	88 791
Added labels per vertex	4.52	5.23	4.50

Table 7.3: The continuation of Table 7.2, showing the results for the remaining four configurations.

Algorithm	CLA			
Potentials?	✗	✗	✓	✓
Pruning?	✓	✓	–	–
Improved dominance?	✗	✓	✗	✓
Time [s]	11.528	9.568	7.937	6.576
Avg. label set size	3.36	2.88	3.25	2.74
Max. label set size	34	14	33	14
Visited vertices	2 378 449	2 378 449	2 170 353	2 170 353
Visited labels	11 374 719	9 859 942	9 971 268	8 398 825
Visited labels per vertex	4.58	4.02	4.26	3.63
Rescanned labels	302 825	47 498	0	0
Added labels per vertex	4.95	4.34	4.64	3.96

a charging station are much simpler: Naive CLA represents information about energy consumption and current SoC with piecewise linear functions, and both operations require a linear sweep over the supporting points of these functions, whereas for CLA, they only need to update the three values of the vertex label.

Without speedups, there is no stopping criterion, so CLA visits all vertices in the graph. Per vertex, it adds 5.23 labels and visits 4.85 labels on average. This implies that the majority of added labels remain in their label sets until they are settled for the first time, while a small portion of labels is dominated and removed before being settled. Because the algorithm is label-correcting, it is possible for labels to be settled more than once. We call these labels *rescanned* labels. Without speedups, approximately 2.7% of all visited labels are rescanned. Using improved dominance criteria decreases the number of added labels by 14%. Additionally, the ratio of rescanned labels decreases to about 0.5%. This decreases the running time by about three seconds.

Table 7.4: This table shows the performance of the PSA preprocessing phase, which computes the shortcut graph by performing one profile query for each charging station in the graph. The first row gives the overall time for computing the entire shortcut graph, while the remaining values are averaged from the individual queries.

Graph	*luxembourg*		*germany*	
Potentials?	×	✓	×	✓
Overall time	803ms	725ms	25m 36s	23m 29s
Time per query [ms]	22	20	798	732
Visited vertices	20 918	20 918	526 577	526 577
Rescanned vertices	1 449	129	14 354	2 694
Avg. profile size	1.26	1.26	1.19	1.19

If the pruning technique is enabled, the number of visited vertices decreases to about 50.7% of the graph on average, which approximately halves the average running time. Potential shifting makes the algorithm label-setting, which makes it impossible for labels to be rescanned. Furthermore, it adds a stopping criterion, which allows the search space to be restricted even further. However, this only results in a slight improvement, reducing the ratio of visited vertices to 46.3%. The reason for this is that the *germany* graph only contains relatively short paths with a negative energy consumption. Therefore, the pruning technique causes the remaining labels to be discarded shortly after the target is reached for the first time.

By enabling potential shifting and improved dominance criteria, CLA achieves its best average running time of 6.6 seconds. This is a speedup of 9.6 compared to Naive CLA and 3.4 compared to CLA without speedups.

7.2.2 Profile Shortcut Algorithm

In this section we evaluate the performance of the Profile Shortcut Algorithm introduced in Chapter 6. We start by evaluating the preprocessing phase of the algorithm, in which the shortcut graph is computed by performing a profile query from every charging station. For this purpose, we ran the preprocessing on both the *luxembourg* graph and the *germany* graph. For the *luxembourg* graph we used a smaller battery of 4kWh, since otherwise the entire graph would be traversable without recharging. On each graph, we performed the preprocessing twice, once with potential shifting and once without. The results are shown in Table 7.4.

On the *luxembourg* graph, the preprocessing is fairly fast. On average, each profile query takes 22 milliseconds without potential shifting. Since there are 37 charging stations in total, this leads to a total running time of 803 milliseconds. Each query stops once the initial battery capacity is depleted. Even with a small 4kWh battery, the queries still visit 57.3% of the vertices in the graph on average, since the graph is so small. Without potential shifting, 6.9% of the visited vertices are rescanned; potential shifting reduces this to 0.6% but cannot eliminate rescanning entirely, since the profile queries remain label-correcting. The size of the computed energy consumption profiles is very low, containing only 1.26 supporting points on average. This indicates that most profiles consist only of a single constant value.

On the *germany* graph, the average running time for each query is much longer, averaging 798 milliseconds without potential shifting. The reason for this is that the increased battery capacity of 16kWh allows a much larger range of vertices to be explored. While the

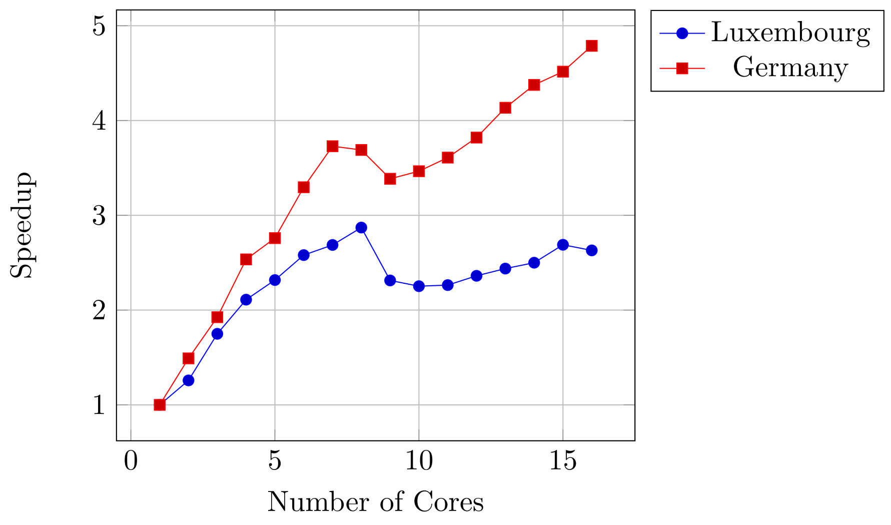

Figure 7.1: This figures shows the speedup for the parallelized preprocessing phase of PSA, depending on the number of cores used.

number of visited vertices is approximately 25 times as large as in the *luxembourg* graph, they still constitute only 11.2% of all vertices in the graph. Only 2.7% of all visited vertices are rescanned without potential shifting; potential shifting reduces this to 0.5%. With 1.19 supporting points per profile, the average profile size is lower than in the *luxembourg* graph, likely because the overall ratio of charging stations is smaller in the *germany* graph.

Because the *germany* graph contains so many charging station, the entire preprocessing phase takes over 23 minutes even with potential shifing. To improve the running time, we parallelize the preprocessing phase: Since the individual profile queries for each charging station are independent of each other, we can easily distribute them among the 16 cores of our test machine. We thus repeat the previous experiment, this time distributing the queries among an increasing number of cores. Since the previous experiment showed that potential shifting improves the running time, we always enable it for this experiment. Figure 7.1 shows the achieved speedup for each number of used cores.

If the preprocessing phase were perfectly parallelizable, we would expect a speedup factor of 16 for 16 cores. As the data shows, the actual speedup factor is 4.8 for the *germany* graph and 2.6 for the *luxembourg* graph. One reason for this discrepancy is that each thread needs its own copies of the vertex labels, to prevent different queries from interfering with each other. The code for copying the vertex labels cannot be parallelized, since every thread needs to perform it. On the *luxembourg* graph, the queries themselves have a very short running time, even when compared to the overall size of the graph. Therefore, copying the vertex labels takes up a larger ratio of the overall time when compared to the *germany* graph, which contributes to the lower speedup factor.

Additionally, we observe that the speedup increases almost monotonically except for the change from 8 to 9 cores, where it decreases momentarily. The reason for this is most likely that the machine on which the experiments were performed has two processors with eight cores each. As long as eight cores or less are used, the queries are all performed by the same processor. Once nine cores or more are used, however, the queries have to distributed among both processors, which leads to an additional coordination overhead.

Even though the speedup factor is lower than expected, parallelization with 16 cores still leads to a significantly improved preprocessing time of 277 milliseconds for the *luxembourg* graph and 4:55 minutes for the *germany* graph.

Table 7.5: This table shows the performance of PSA with various combinations of speedup techniques. For this experiment, the same 1000 random queries on the *germany* graph were evaluated for each configuration of the algorithm.

	Potentials?	×	×	✓	✓
	Trimmed profiles?	×	✓	×	✓
Preprocessing	Parallelized time [m:ss]	5:02	5:14	4:51	4:55
	Time per query [s]	1.732	1.816	1.728	1.751
Initial query	Time [ms]	396	390	352	352
	Visited vertices	478 721	478 721	445 664	445 664
	Rescanned vertices	13 238	13 238	0	0
Target processing	Time [ms]	674	689	675	684
	Visited vertices	420 606	420 590	420 606	420 590
	Rescanned vertices	19 225	19 091	3 903	3 736
Shortcut query	Time [ms]	56	56	35	33
	Visited vertices	1 717	1 717	871	871
Overall	Total query time [s]	1.126	1.135	1.062	1.070
	Avg. profile size	1.21	1.19	1.21	1.19

We continue by evaluating the performance of the query part of PSA. In addition to the basic algorithm, we developed two speedup techniques: Potential shifting was applied to all of the four phases that make up PSA. This introduced a stopping criterion for the initial query from the source vertex and the query on the shortcut graph, as well as reducing the number of rescanned vertices in the other phases. Additionally, we introduced a method for trimming the energy consumption profiles, making them monotonically increasing. We tested the performance of PSA as well as its speedup techniques by evaluating the same randomly chosen 1000 queries on the *germany* graph for every possible combination of speedups. The results are shown in Table 7.5.

Without speedups, the overall running time of an average query is 1.126 seconds. This is distributed among the three phases of the query as follows: The initial query from the source vertex takes 392 milliseconds (35%), adding the target vertex to the shortcut graph takes 674 milliseconds (60%), and the query on the shortcut graph takes 56 milliseconds (5%). We observe that the query on the shortcut graph is much faster than the other two phases. The reason for this is that the first two phases scan all vertices that are reachable from the source or target vertex with a 16kWh battery, while the shortcut graph query only scans the charging stations in the graph, of which there are much fewer. If a larger graph with more charging stations or a smaller battery capacity is used, the share of the shortcut query in the overall running time will increase.

When examining the running time per settled vertex, we observe that the shortcut query is much slower: The initial query and target processing query settle 1253 and 594 vertices per milliseconds, respectively, while the shortcut query processes only 28. There are two reasons for this: One is that when relaxing an edge, the shortcut query has to scan the profile of the query to find the optimal SoC. The other reason is that the density of edges is much higher in the shortcut graph than in the original graph, since there is an edge between every pair of charging stations that are reachable from each other. Therefore, more edges are relaxed per vertex in the shortcut graph.

We observe that trimming the profiles has only a miniscule effect on their average size, decreasing it from 1.21 to 1.19 supporting points. This is not surprising, since the majority of profiles already consist of only one supporting and thus cannot be trimmed further. Since the trimming operation itself is linear in the size of the profiles, it slightly increases the running time of the preprocessing and target processing phases, where the profiles are calculated. The resulting speedup in the other two phases cannot make up for this, leading to a slightly increased overall running time.

Even without potential shifting, the overall ratio of rescanned vertices is low, averaging 2.8% for the initial query and 4.6% for the target processing phase. Potential shifting decreases this to none for the initial query and 0.9% for the target processing phase. Additionally, it adds a stopping criterion to the initial query and the shortcut query. In the initial query, this stopping criterion only takes effect if the target is reached directly, which is rare for the given input data. Therefore, the number of visited vertices decreases only slightly. The effect is more dramatic for the shortcut query, where the number of visited vertices is approximately halved. Overall, potential shifting leads to a slightly improved overall running time of 1.062 seconds without profile trimming and 1.070 seconds with profile trimming.

Overall, we observe that the best performance is achieved when enabling potential shifting and parallelizing the preprocessing phase. This leads to a preprocessing time of slightly below 5 minutes and a query time of slightly over a second. Trimming the profiles increases the running times slightly, but the effect is very small.

7.2.3 Comparison

To conclude our evaluation, we compare the performance of the two algorithms, now with speedup techniques enabled. First we compare the overall running time for the same set of randomly chosen queries. Then we analyse the impact of the initial SoC at the source vertex and the overall ratio of battery swapping stations in the graph. Finally, we examine the behavior of the running time for increasing query distances.

Overall Running Time. To analyse the overall performance, we evaluated the same randomly chosen 10000 queries on the *germany* graph for both algorithms. The results are shown in Tabless 7.6 and 7.7.

We observe that the query performance of PSA is vastly superior to that of CLA. There are two main reasons for this: The first reason is that the average search space of PSA is much smaller. The original graph is only used to find the path from the source to the first charging station and from the last charging station to the target, while the remainder of the search is conducted on the much smaller shortcut graph. The searches on the original graph are stopped as soon as the initial battery capacity is exhausted, so the overall search space is mainly determined by the battery capacity rather than the distance between the source and the target, as is the case for CLA.

The second reason is that CLA uses Pareto sets to handle charging stops, which causes it to settle an average of 3.70 labels per vertex, while PSA only settles each vertex once (save for a very small number of rescanned vertices). While PSA uses energy consumption profiles in several phases, the complexity of these profiles is very limited, averaging only 1.19 supporting points. The major performance downside of PSA, however, is that it requires a fairly long preprocessing phase that takes approximately 23 minutes on a single core and 5 minutes on 16 cores. This preprocessing phase has to be performed once per graph and vehicle. Therefore, the use of PSA is only feasible if a moderately large number of queries is expected to be performed for a particular vehicle on a particular graph.

Impact of Initial SoC. To evaluate the impact of the initial SoC, we evaluate 100 randomly chosen queries for CLA and 1000 randomly chosen queries for PSA on the *germany* graph. We evaluate these queries ten times each, starting with an initial SoC of 10% and increasing it in 10% steps each time. The results are shown in Figure 7.2.

Table 7.6: This table shows the performance of CLA for 10000 random queries on the *germany* graph. The performance of PSA for the same 10000 queries is shown in Table 7.7.

Charging Label Algorithm	
Time [s]	7.992
Avg. label set size	2.77
Max. label set size	14
Visited vertices	2 342 145
Visited labels	9 184 331
Visited labels per vertex	3.70
Added labels per vertex	4.29

Table 7.7: This table shows the performance of PSA for 10000 random queries on the *germany* graph. The performance of CLA for the same 10000 queries is shown in Table 7.6.

Profile Shortcut Algorithm		
Preprocessing	Parallelized time [m:ss]	4:53
	Time per query [s]	1.697
	Visited vertices	526 577
	Rescanned vertices	2 694
Initial query	Time [ms]	371
	Visited vertices	452 915
Target processing	Time [ms]	706
	Visited vertices	422 559
	Rescanned vertices	3 756
Shortcut query	Time [ms]	24
	Visited vertices	882
Overall	Total query time [s]	1.101
	Avg. profile size	1.19
	Max. profile size	16
	Visited vertices	876 356

For CLA, we observe that the running time is much lower for a SoC of 10%, averaging 4.5 seconds. The running time quickly increases for higher SoCs, and from 30% upwards no general trend is observable. This can be explained by the fact that for a very low initial SoC, no nearby charging stations may be reachable, making it impossible to reach the target. In this case, the query stops early, lowering the average running time. Once the initial SoC is high enough to reach the first charging station, however, it has no further impact.

For PSA, we observe a slight increase in running time with increasing initial SoC, from 0.82 seconds for an initial SoC of 10% to 1.07 seconds for an initial SoC of 100%. This is because the initial SoC directly impacts the size of the search space for the initial query: Since there is no stopping criterion, the query will run until the initial SoC is depleted. For a higher SoC, the range of reachable vertices increases, thereby also increasing the running time. The average running time for the target processing phase, however, decreases. This can be explained by the fact that for a higher initial SoC, it is more likely that the vehicle can reach the target without using a charging station. In this case the algorithm will finish before the target processing phase. Since the increase in the running time of the initial

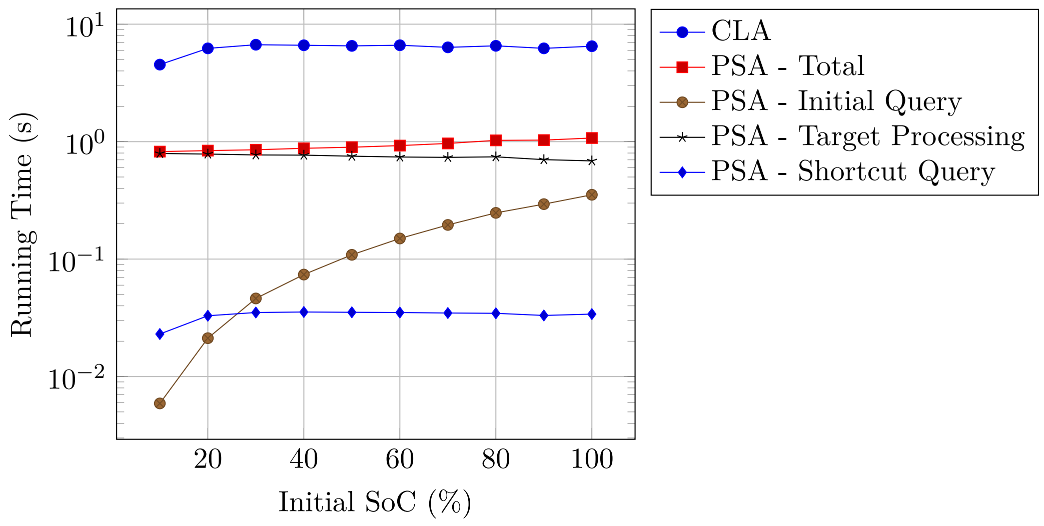

Figure 7.2: Average running times of CLA and PSA, as well as the various phases of PSA, depending on the initial SoC of the vehicle.

query is more dramatic than the decrease in the running time for the target processing phase, the overall running time increase with the initial SoC.

Impact of Battery Swapping Stations. So far, we used the same distribution of charging stations for all experiments. To examine the effect that battery swapping stations have on the performance of our algorithms, we perform a series of queries, gradually increasing the ratio of battery swapping stations. We start with only regular charging stations, evaluating 100 randomly chosen queries for CLA (once with and once without improved dominance criteria) and 1000 randomly chosen queries for PSA on the *germany* graph. We repeat these queries ten times, each time increasing the ratio of battery swapping stations in the graph by 10%, until only battery swapping stations are used. The results are shown in Figure 7.3.

We observe that the running time of CLA without improved dominance criteria increases dramatically as the ratio of battery swapping stations increases, from 6.5 seconds on average with no battery swapping stations to 37.7 seconds with only battery swapping stations, whereas the running times of CLA with improved dominance criteria and PSA both stay approximately constant. This can be explained by the fact that CLA handles battery swapping stations differently than regular charging stations: When encountering a battery swapping station while relaxing an edge, a duplicate of the current label is made that represents swapping the battery at this station.

Without improved dominance criteria, neither of these two labels dominates the other: The duplicated label has a higher maxSoC value, allowing the vehicle to drive farther without having to recharge again. The sum cons + minSoC, however, is lower for the original label. This criterion represents the lowest possible energy consumption that can be reached in the further course of the path, so if overcharging occurs, the energy consumption value of the duplicated label may rise above that of the original label. Since both labels are retained, the overall number of labels in the graph increases with each additional battery swapping station. Thus, the average label set size increases from 2.8 for no battery swapping stations to 11.6 for only battery swapping stations, which explains the increased running time.

If improved dominance criteria are used, the second dominance criterion, which was previously cons + minSoC, is modified to take into account the maximum recuperation that is possible from the current vertex. Because there are relatively few edges with a

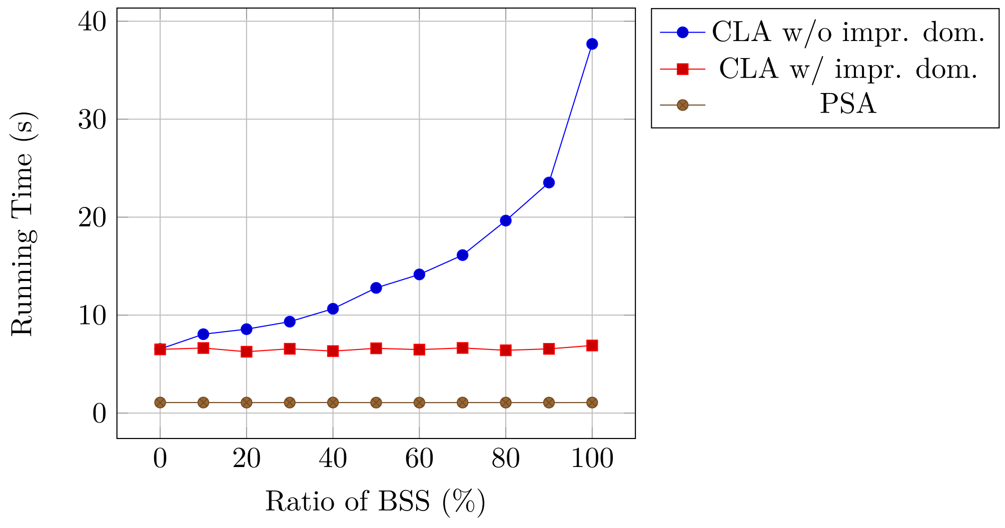

Figure 7.3: Average running times of CLA (with and without improved dominance criteria) and PSA depending on the ratio of battery swapping stations in the graph. The remaining charging stations are regular charging stations.

negative energy consumption value in the graph, the maximum recuperation possible at many vertices will be 0. This means that the energy consumption value cannot sink below its current value and that the second dominance criterion is equivalent to cons. Since cons has the same value for both the original label and its duplicate, the only remaining dominance criterion in which they differ is maxSoC, where the duplicated label is superior. Therefore, it dominates the original label. Since this situation occurs at most battery swapping stations, the overall number of labels in the graph does not increase significantly, which explains the approximately constant running time for CLA with improved dominance criteria.

For PSA, we also observe that the ratio of battery swapping stations does not significantly affect the running time. This is not surprising, since the algorithm requires no special handling for battery swapping stations. In fact, the shortcut query becomes slightly faster with an increasing ratio of battery swapping stations, since deciding how much to charge does not require a linear sweep over the supporting points of the profile function. However, since most profiles have only one supporting point and the shortcut query only constitutes a small fraction of the overall running time, this effect is negligible.

Impact of Query Distance. Now we want to analyze the impact of the query distance on the performance of our algorithms. In order to measure the distance of a query, we use the *Dijkstra rank*. In a label-setting version of Dijkstra's algorithm, each vertex is settled only once. Therefore, the order in which the vertices are settled is well-defined. Given a fixed source vertex s, we run a single-source Dijkstra query from s, using energy consumption as the edge weight and applying potential shifting to make the algorithm label-setting. We do not take battery constraints into account. The Dijkstra rank of a vertex v is then defined as the number of vertices that were settled before v.

Using Dijkstra rank, we perform the following experiment: For every $0 \leq i \leq 22$, we randomly choose 100 queries for CLA and 1000 queries for PSA on the *germany* graph such that for each query the Dijkstra rank of the target vertex as measured in relation to the source vertex is 2^i. This way, the distance of the query as measured by the Dijkstra rank doubles with each step. Since a Dijkstra rank of 2^{23} is not possible in our graph, we stop the experiment at this point. The results of the experiment are shown in Figure 7.4.

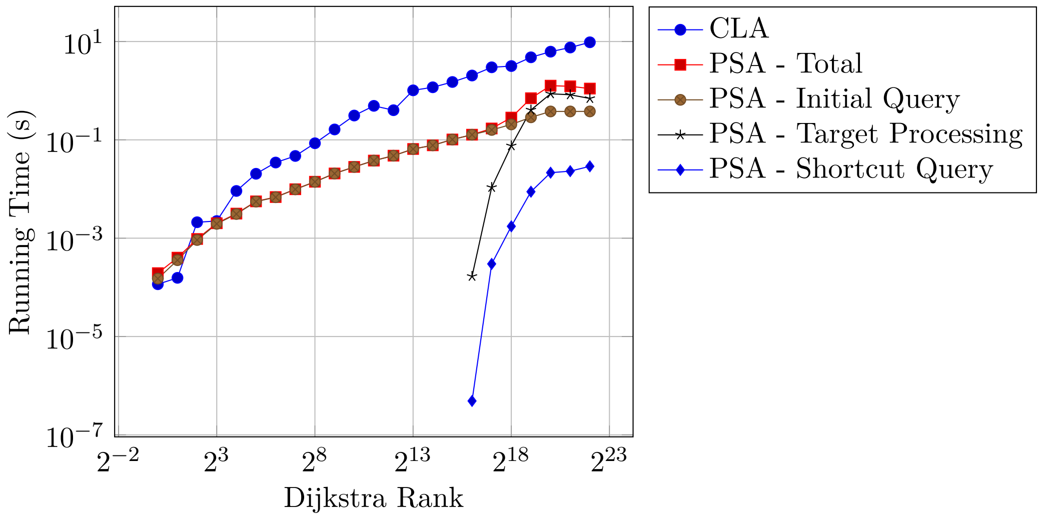

Figure 7.4: Average running times of CLA and PSA, as well as the various phases of PSA, depending on the Dijkstra rank of the query.

We observe that the running time of both algorithms increases with the Dijkstra rank of the query. For low ranks, the performance of both algorithms is approximately equal. However, as the rank increases, the label sets used by CLA start to become larger, making it slower than PSA. From this point onwards, the running time of CLA rises fairly steadily. The running time of PSA also rises steadily until a rank of approximately 2^{15}. Up until this point, the initial SoC of 16kWh always suffices to reach the target, so only the initial query is performed. For a rank of approximately 2^{17} to 2^{20}, we observe a sharp rise in the running time of PSA. At this point, it becomes increasibly common that the vehicle requires recharging to reach the target, in which case the target processing query and shortcut query must be performed. As a result, the average running times of these queries rise sharply, and starting at a rank of 2^{19}, the target processing query starts to dominate the running time.

At a rank of 2^{20}, the overall running time becomes mostly stable. At this point, the target is always too far away from the source to be reached without recharging, so the target processing query and shortcut query are always performed. Since the running time of the target processing query depends only on the position of t and not that of s, it remains approximately constant from now on. Likewise, the initial query now always runs until the entire battery is depleted, so its running time also becomes constant. Only the running time of the shortcut query on the shortcut graph may still increase with the rank, since more charging stops become necessary with increasing distance. However, since the shortcut graph for the *germany* graph contains fewer than 2000 vertices, the running time of the shortcut query contributes only a small amount to the overall running time, which therefore remains almost contant. In larger graphs with more charging stations, the shortcut query has a larger contribution to the overall running time and may eventually start to dominate it for high ranks.

In summary, we conclude that the performance of PSA is superior to that of CLA in most respects. Using the Peugeot iOn model with a battery capacity of 16kWh on the *germany* graph, its query times are approximately seven times faster on average than those of CLA. The performance of both algorithms is independent of the ratio of battery swapping stations in the graph, although improved dominance criteria have to be used for CLA to achieve this. Both algorithms perform slightly better for a lower initial SoC, but the PSA is always several times faster than CLA for any initial SoC. The downside of PSA is that it requires a preprocessing phase that takes approximately 23 minutes when using a single core. The preprocessing phase can be parallelized, improving the running time to approximately 5 minutes for 16 cores. Thus, while PSA is superior when performing many queries for a fixed combination of input graph and used vehicle, CLA is superior if the graph or the used vehicle are expected to change every few queries.

8. Conclusion

To conclude this thesis, we give a short summary of our results as well as an outlook on possible future work.

8.1 Summary

In this thesis, we formulated the Energy-Optimal Electric Vehicle Routing with Recharging problem and developed two algorithms for solving it. After giving a precise problem statement in Chapter 3, we introduced the first algorithm in Chapter 4, which we called the Charging Label Algorithm (CLA). We began by modelling the charging process with piecewise linear functions, which we then showed could be simplified into vertex labels consisting of three static values. Using these labels, we developed a variation of the multi-objective shortest path search algorithm which uses Pareto sets to represent all tentatively optimal solution.

In Chapter 5, we analyzed the complexity of CLA by studying the sizes of the involved Pareto sets. We presented an example graph which generates Pareto sets of exponential size for the multi-objective shortest path search algorithm. By analyzing the situations in which vertex labels are dominated, we were able to prove that the Pareto sets of CLA are polynomial for chain graphs, a class of graphs which we defined as generalization of the example graph, under the condition that the graphs only contain regular charging stations. This yields an overall worst-case complexity of $\mathcal{O}(m^5)$ for CLA.

In Chapter 6, we developed an alternative algorithm which uses profile queries to precalculate the shortest paths between all pairs of charging stations. This information is then used to speed up the queries. We called this algorithm the Profile Shortcut Algorithm (PSA).

We evaluated the performance of both algorithms on the road network of Germany in Chapter 7, observing average query times of 8 seconds for CLA and 1 second for PSA. While the query times of PSA are significantly faster than that of CLA, it requires a long preprocessing phase of 23 minutes. By parallelizing the preprocessing phase, it can be shortened to 5 minutes on 16 cores.

8.2 Future Work

While we were able to develop two algorithms for solving the EEVRC problem with fairly short query times even on large graphs, several open questions and possibilities for further improvement still remain.

To further improve the running times of the two algorithms, well-known speedup techniques such as Contraction Hierarchies and A* Search could be adapted. When applied to similar algorithms such as the ones developed in [Zün14] and [EFS11], these techniques resulted in vastly improved running times, making the algorithms feasible for interactive applications even on large graphs like the road network of Europe.

Another possible approach for improving the performance of CLA could be to employ heuristics in order to reduce the size of the used Pareto sets. One such heuristic could be to assume that the battery is always charged to its full capacity at every charging station, which would eliminate the need for considering entire charging intervals. While this can lead to overcharging in the further course of the path in some cases, the results of our experiments suggest that these cases are rare and only result in a slightly increased overall energy consumption.

While we were able to prove the polynomiality of CLA for chain graphs with regular charging stations, it is unclear if this holds true for all road network graphs or for other charging station types. Further research could focus on extending our proof to more general graph classes by showing that they can be reduced to chain graphs. Regarding other charging station types, the results of our evaluation suggest that the addition of superchargers does not increase the size of the Pareto sets. While we observed increasing Pareto set sizes when adding battery swapping stations, we were able to show that each battery swapping station can add at most one label to the Pareto set of its corresponding vertex.

For PSA, we observed that the complexity of the algorithm depends on the size of the calculated energy consumption profiles. Our evaluation data suggests that these profiles are very small in practice, but it is unclear if their worst-case size is always polynomial in the size of the graph.

Bibliography

[AHLS10] Andreas Artmeier, Julian Haselmayr, Martin Leucker, and Martin Sachenbacher. The Shortest Path Problem Revisited: Optimal Routing for Electric Vehicles. In *Proceedings of the 33rd Annual German Conference on Advances in Artificial Intelligence*, volume 6359 of *Lecture Notes in Computer Science*, pages 309–316. Springer, September 2010.

[BDPW13] Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. Energy-Optimal Routes for Electric Vehicles. Technical Report 2013-06, Faculty of Informatics, Karlsruhe Institute of Technology, 2013.

[DGPW11] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable Route Planning. In *Proceedings of the 10th International Symposium on Experimental Algorithms (SEA'11)*, volume 6630 of *Lecture Notes in Computer Science*, pages 376–387. Springer, 2011.

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

[EFS11] Jochen Eisner, Stefan Funke, and Sabine Storandt. Optimal Route Planning for Electric Vehicles in Large Network. In *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence*. AAAI Press, August 2011.

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

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability. A Guide to the Theory of $\mathcal{NP}$-Completeness.* W. H. Freeman and Company, 1979.

[GSSD08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, June 2008.

[HRZL09] Stefan Hausberger, Martin Rexeis, Michael Zallinger, and Raphael Luz. Emission Factors from the Model PHEM for the HBEFA Version 3. Technical Report I-20/2009, University of Technology, Graz, 2009.

[Joh77] Donald B. Johnson. Efficient Algorithms for Shortest Paths in Sparse Networks. *Journal of the ACM*, 24(1):1–13, January 1977.

[Mar84] Ernesto Queiros Martins. On a Multicriteria Shortest Path Problem. *European Journal of Operational Research*, 26(3):236–245, 1984.

[SF12] Sabine Storandt and Stefan Funke. Cruising with a Battery-Powered Vehicle and Not Getting Stranded. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*. AAAI Press, 2012.

[Zün14] Tobias Zündorf. Electric Vehicle Routing with Realistic Charging Models. Master thesis, Karlsruhe Institute of Technology, November 2014.