Karlsruhe Institute of Technology

Consumption and Travel Time Profiles in Electric Vehicle Routing

Master Thesis of

Simeon Andreev

At the Department of Informatics
Institute of Theoretical Computer Science

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

Time Period: 1st November 2014 – 30th April 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, 30th April 2015

Abstract

Electric vehicles introduce new problems in vehicle route planning. Due to limited battery capacity the fastest route from origin to destination is not always feasible. Existing techniques compute the energy optimal route, however this route is often too conservative in terms of travel time. Optimizing both energy consumption and travel time may allow us to identify routes which are both fast and energy efficient. In this thesis we examine the *tradeoff* between consumption and travel time in the form of profile functions. We acknowledge the fact that an electric vehicle may be driven slower to spare energy and so allow variable driving speeds on street segments. In a recent publication, Baum et al. offer a Pareto optimization approach which deals with the same problem. In our evaluation we show that the profile algorithm we develop outperforms this recent approach.

Deutsche Zusammenfassung

Sowohl der Energieverbrauch als auch die Reisezeit sind in der Routenplanung für Elektroautos von Bedeutung. Frühere Techniken berechnen entweder die schnellste oder die energie-effizienteste Route, wobei erstere in der Regel zu viel Energie und letztere zu viel Zeit kostet. Der Pareto-Ansatz von Baum et al. optimiert sowohl die Reisezeit als auch den Verbrauch, um den Trade-off zwischen diesen zwei Kriterien zu berechnen. Dabei werden auch variable Fahrgeschwindigkeiten auf Straßensegmenten betrachtet: das Elektroauto darf langsamer fahren, um Energie zu sparen. In dieser Masterarbeit entwerfen wir eine andere Modellierung für das gleiche Problem. Mit Hilfe von Funktionen beantworten wir Profilanfragen, die sowohl schneller als auch präziser sind im Vergleich zu dem Ansatz von Baum et al.

Contents

1. Introduction

The focus of standard vehicle route planing is to offer drivers the fastest route from origin to destination. Time-dependent routing addresses the fact that fastest routes may vary due to congestions and rush hours. In both the standard and the time-dependent scenario street segments are traversed at top speed, where in the latter scenario top possible speeds vary based on the day of the week and the time of the day. With the introduction of electric vehicles, however, the notion of allowing different speeds on the same street segment becomes important.

Driving at lower speeds in general leads to lower consumption, regardless of whether the vehicle is electric or not. This fact is ignored in standard routing, as gas stations are readily available and the cruising range of standard vehicles is extensive. Electric vehicles, on the other hand, introduce the range anxiety: the cruising range is limited due to comparatively low battery capacities and sparse charging stations. Driving at top speed may hinder even the most energy efficient route from origin to destination, and driving at low speed is too conservative in terms of travel time. Variable driving speeds at street segments permit a compromise between consumption and travel time. In other words, a consumption and travel time tradeoff exists which may be utilized in a number of ways.

Given an origin and a destination, techniques exist which compute the earliest arrival route or the most energy efficient route which an electric vehicle can traverse. Both routes are offered by the full consumption and travel time tradeoff, and with greater precision as more driving speeds are taken into account. The tradeoff may also offer sweet spots, routes which are both fast and energy efficient.

The sparsity of charging stations has led to recent techniques which explicitly integrate charging [Sto12, Zün14]. In such approaches the driver is guided from one charging station to the next, until the destination is reached in optimal time. As in standard routing, these techniques assume top speed at segments. However, by driving at a lower speed the driver may reach an otherwise unreachable charging station and continue on a fast route. As it is presumed unreachable, such a route is discarded by the existing techniques – although it may be the optimal route in terms of travel time.

In this thesis, we explore models and algorithms required to compute piecewise linear functions that represent the consumption and travel time tradeoff based on variable driving speed. Specifically, we model the street network as a weighted graph with functions as edge costs. We then adapt algorithms and speed-up techniques from the time-dependent route planing to compute consumption and travel time profiles.

1.1. Related Work

Numerous works exist on topics in standard, time-dependent and electric vehicle routing. In almost all such publications, some variant of the shortest path problem is solved with an adaptation of Dijkstra's algorithm [Dij59]. In a multi-criteria setting, e.g. minimizing travel time and consumption simultaneously, Dijkstra may be combined with Pareto optimization [CM85]. The A* search [HNR68b] is one of the earlier improvements of Dijkstra's algorithm, and is often applied when optimizing multiple criteria. Dijkstra's search is guided by the A* goal direction, in order to reduce the search space size. Another method of reducing search spaces in a general setting is applying shortcuts, which allow Dijkstra's algorithm to skip multiple hops. One such method is the Contraction Hierarchies (CH) algorithm [GSSD08], which makes use of the hierarchy found in street networks. Due to the massive speed-up of CH over Dijkstra in standard routing, the approach is adapted in many other scenarios [BGNS10, EFS11, DGNW13, HF14, DSW14, Zün14]. Customizable Route Planning (CRP) [DGPW11] is another algorithm which takes advantage of shortcuts. This technique utilizes partitioning and shortcuts to skip over whole partitions of the graph during the Dijkstra search.

Time-Dependent Routing. Dynamic scenarios when computing shortest paths is first considered by Orda et al. in [OR90]. Their work is applied in time-dependent routing, where traffic jams and rush hours cause increased travel time on street segments during specific times of the day. According to Kaufmann et al. [KS93] Dijkstra's algorithm answers *earliest arrival* queries in polynomial time, assuming the street segments of the network satisfy a FIFO property. This property ensures that waiting to traverse a segment at a later time is not beneficial. A time-dependent *profile* consists of earlier arrival times for all possible departure times of a day. Time-dependent profiles are the basis for the consumption and travel time profiles of this thesis.

Batz et al. adapt the CH algorithm to answer both earliest arrival queries and profile queries in a time-dependent setting, [BGNS10]. They utilize profiles during the hierarchy construction, due to the fact that arrival time is unknown at shortcut creation. To answer an earliest arrival query, Batz et al. adjust the query to run an upward and a downward phase, as opposed to the standard forward and backward search of the CH algorithm. Furthermore, Batz et al. make use of function simplification and edge pruning to allow profile queries running in a second on the scale of continental networks. Along with the A* technique, the approximated time-dependent CH of Batz et al. is the speed-up technique we choose to apply in this thesis.

Electric Vehicle Routing. Energy efficient routes become important with the introduction of electric vehicles. With Johnson's potential shifting [Joh73] and Dijkstra's algorithm, Eisner et al. [EFS11] are able to compute energy-optimal routes. They model battery constraints and energy recuperation, to account for the specifics of electric vehicles. Moreover they adapt the CH algorithm to speed up their technique, sinking query times to the scale of milliseconds in continental sized networks.

By using the same modelling of the electric vehicles battery, but a more thorough consumption model, Baum et al. also compute energy-optimal routes in [BDPW13]. To improve the scalability of their approach they adapt CRP, offering preprocessing times in seconds again on continental scale. This enables flexible updates of the consumption costs on street segments.

Earliest arrival times in EV routing are first considered by Storandt in [Sto12]. Storandt solves Constrained Shortest Path (CSP) problem, where the battery constraints restrict available routes. An adaptation of the CH algorithm is also presented, yielding query times in milliseconds. Furthermore, Storandt considers charging stations which are randomly

distributed in the street network and vastly increase the cruising range. Throughout, only static travel times on street segments are considered.

In a more recent work, [Zün14], Zündorf also explores the computation of earliest arrival times with the integration of charging stations. In contrast to [Sto12], Zündorf considers the time required by battery charging and uses actual locations of charging stations throughout Europe. In addition, different types of charging stations are handled in [Zün14] and a further adaptation of CH is presented. However, travel times on segments remain static.

Hartmann and Funke compute routes which are both energy efficient and fast in [HF14], by solving CSP. They incorporate varying driving speeds on street segments to some extent and also adapt CH. However, they do not respect battery constraints, as their approach focuses on standard vehicles. Furthermore, they apply heuristics to achieve running times in seconds on a Germany sized country scale.

Baum et al. [BDHS+14] examine variable speeds in EV routing, to compute tradeoffs between travel time and energy consumption. Their approach utilizes a multi-criteria search which considers different driving speeds on street segments and obeys battery constraints. However, they only allow a fixed (segment specific) number of different speeds on a segment. As in [HF14], Baum et al. also apply heuristics, however in doing so they achieve queries on the scale of seconds for continental sized networks. They do not consider speed-up techniques which requires preprocessing.

An approach which also computes consumption and travel time tradeoffs is introduced by Goodrich and Pszona in [GP14]. They utilize a two-phase bicriterion search which finds fast prefix paths and an energy efficient suffix ones, and then combines prefixes and suffixes. Goodrich and Pszona allow three different driving speeds at each street segment. They too incorporate randomly placed charging stations as in [Sto12], however they allow only a few stations (1 to 15) and charging stations may only charge the battery to full. Furthermore, the authors do not examine the application of speed-up techniques. The running times of their heuristic approach are on the scale of seconds for a country sized street network.

1.2. Contribution and Outline

The contributions of this thesis are as follows:

- a street network model that allows the traversal of street segments with any travel time within a segment specific interval
- a definition of a consumption and travel time tradeoff function
- an adaptation of the time-dependent linking operator for tradeoff functions and an algorithm which (efficiently) computes the link result of two tradeoff functions
- an adaptation of the time-dependent earliest arrival profile algorithm for the computation of consumption and travel time profiles
- an adaptation of the time-dependent CH algorithm for the computation of consumption and travel time profiles
- an extensive experimental evaluation of all introduced approaches

The rest of this thesis is structured as follows:

In Chapter 2 we offer a more thorough introduction into electric vehicle routing and techniques used in route planing which are relevant for this thesis.

In Chapter 3 we discuss the modelling of our profile functions, as well as the core function operations we require for our algorithms. These are the link operation, the merge operation and the domination test.

In Chapter 4 we define our basic algorithms, which compute the consumption and travel time profiles.

In Chapter 5 we examine the application of the A* algorithm and the adaptation of the time-dependent variant of the CH algorithm.

In Chapter 6 we conduct the experimental evaluation of our basic algorithms and the speed-up techniques we use.

In Chapter 7 we offer a summary of this thesis as well as an outlook of possible future work.

2. Preliminaries

We begin the preliminaries with fundamental mathematical constructs which we require throughout the thesis. We then move to graph theory and shortest path algorithms. Furthermore, we introduce basic concepts used in time-dependent routing and electric vehicle routing. We conclude the chapter with the Contraction Hierarchies algorithm and its time-dependent adaptation.

2.1. Mathematics

Definition 2.1. *Real Function*

A function $f : X \to Y$ is real*, if both the domain X and the codomain Y are subsets of $\mathbb{R}$.*

We denote the set of real functions with $\mathcal{F}(\mathbb{R}, \mathbb{R})$.

Definition 2.2. *Convex and Concave Functions*

A function $f : \mathbb{R} \to \mathbb{R}$ is convex*, if the following condition holds:*

$$\forall x_1, x_2 \in \mathbb{R}, \forall \alpha \in [0, 1] : f(\alpha x_1 + (1 - \alpha)x_2) \leq \alpha f(x_1) + (1 - \alpha) f(x_2)$$

A function $f : \mathbb{R} \to \mathbb{R}$ is concave*, if $-f$ is convex.*

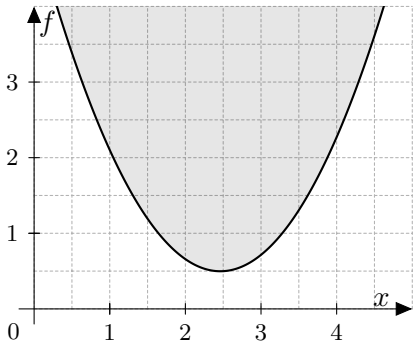
Connecting any two points of a convex function yields a segment that is above the function. Figure 2.1 on the following page shows an example of both a convex and a non-convex function.

Definition 2.3. *Multiset*

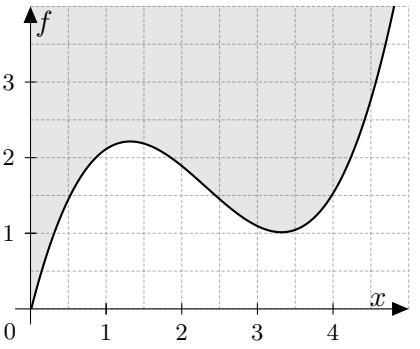
A multiset *is a set which may contain an element multiple times.*

Definition 2.4. *Piecewise Linear Function*

A piecewise linear function *$f : \mathbb{R} \to \mathbb{R}$ is defined by a set of interpolation points in $\mathbb{R}^2$: $\{(x_i, y_i) \in \mathbb{R}^2 \mid i \in 1, \ldots, n\}$, with $\forall i \leq j : x_i \leq x_j$. The value of f at $x \in \mathbb{R}$ is then defined as:*



(a) A convex function.



(b) A non-convex function.

Figure 2.1.: An example of a convex and a non-convex function. The grey area represents the set of points above the respective function. Per definition, if the function is convex, no segment connecting any two function points may lie outside the grey area.

$$f(x) = \begin{cases} \infty & x < x_1, \\ y_i + \lambda_i(x - x_i) & x \in [x_i, x_{i+1}), \\ y_n & \textit{otherwise} \end{cases}$$

where $i \in 1, \ldots, n-1$ *and* $\lambda_i = \frac{y_{i+1}-y_i}{x_{i+1}-x_i}$.

In other words, we set $f(x_i) = y_i$ and for $x \in [x_i, x_{i+1}]$ we interpolate between y_i and y_{i+1}. As f has n interpolation points, we say that the size of f is n. For an example of a piecewise linear function, see Figure 2.2 on the next page.

We define a piecewise linear function f with interpolation points $(x_1, y_1), \ldots, (x_n, y_n)$ as convex, if the interval $[x_1, x_n]$ of f is convex. We use an analogous definition for concave piecewise linear functions.

Let f be a piecewise linear function with interpolation points $\{(x_1, y_1), \ldots, (x_n, y_n)\}$. We define the *slope* of $(x, y) : x \in [x_i, x_{i+1}), y = f(x), i \in 1, \ldots, n-1$ as $m := \frac{y_i - y_{i+1}}{x_y - x_{i+1}}$. For points on f at $x \geq x_n$ we set the slope to 0. If f is also convex and $y_1 \geq y_n$, then $\frac{y_{i-1}-y_i}{x_{i-1}-x_i} \leq \frac{y_i - y_{i+1}}{x_i - x_{i+1}}, i \in 2, \ldots, n-1$ holds. In other words, the slope of f is monotonically increasing.

Definition 2.5. *Parallelogram*

A parallelogram *is a quadrilateral (polygon with four points) with two pairs of parallel sides.*

Definition 2.6. *Dominance*

A point $x = (x_1, \ldots, x_k)^T \in \mathbb{R}^k$ dominates $y = (y_1, \ldots, y_k)^T \in \mathbb{R}^k$*, or* $x \propto y$*, iff both* $\forall i : x_i \leq y_i$ *and* $\exists i : x_i < y_i$ *hold.*

A real function f *dominates another real function* g*, or* $f \propto g$*, iff* $\forall x \in \mathbb{R} : f(x) \leq g(x)$*. If* $\forall x \in \mathbb{R} : f(x) < g(x)$ *also holds, then* f *dominates* g strictly*.*

We denote non-domination by $\not\propto$.

Definition 2.7. *Pareto Set*

A set $X \subseteq Y$ *is a* Pareto set*, if* $\forall x \in X : \{y \in X \mid y \propto x\} = \emptyset$ *holds.*

A point $x \in Y$ *is minimal w.r.t. a Pareto set* X *iff* $x \in X$*.*

In other words, no point in a Pareto set dominates another point in the set.

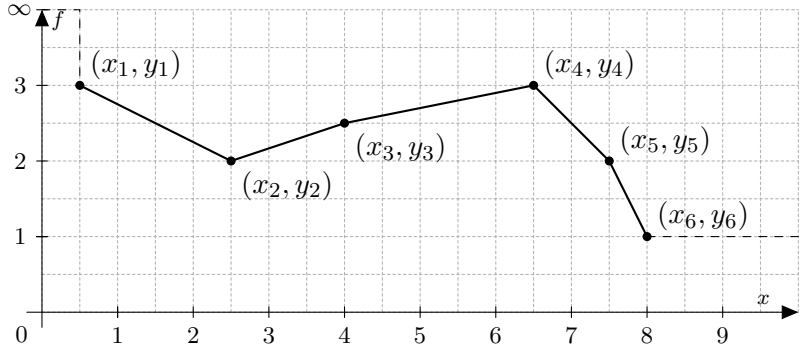


Figure 2.2.: An example of a piecewise linear function. For $x < x_1$ the function value is set to ∞ and for $x > x_6$ the value is set to y_6. In between the function is defined as the linear interpolation between its interpolation points.

2.2. Graph Theory

Graph. A *graph* G is defined as the tuple (V, E), where V is the set of *vertices* and $E \subseteq V \times V$ is the set of *edges*. We say that G is undirected, if $(u, v) \in E \implies (v, u) \in E$ holds. Otherwise, G is directed. A graph $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ iff $V' \subseteq V$ and $E' \subseteq E$.

We say that G is weighted, if the edges of G are weighted by a *cost function* $c : E \to X$. We only examine weighted graphs where the codomain of the cost function lies in $\{\mathbb{R}, \mathbb{R}_+, \mathbb{R}^k, \mathcal{F}(\mathbb{R}, \mathbb{R})\}$.

The reverse graph of G is defined as $G_\mathrm{r} := (V, E_\mathrm{r})$, where $E_\mathrm{r} := \{(v, u) \mid (u, v) \in E\}$. If G is weighted by c, the cost function of G_r is defined as $c_\mathrm{r}((v, u)) := c((u, v)), (v, u) \in E_\mathrm{r}$.

We denote the number of vertices in G as $n := |V|$, and the number of edges in G as $m := |E|$. The neighbourhood of $v \in V$ is defined as $N(v) := \{u \in V \mid (v, u) \in E \cup (u, v) \in E\}$. The node degree of v is the cardinality of $N(v)$, denoted by $deg(v)$. For an edge $e = (u, v) \in E$ we say that u and v are the *head* resp. *tail* vertex of e. At the vertex u we say that e is an outgoing edge, and at v we say that e is ingoing. And last, e is a *loop* if $u = v$.

Definition 2.8. *Path*

A path *P in $G = (V, E)$ is a sequence $(v_1, \ldots, v_n)$ of vertices in G, where $(v_i, v_{i+1}) \in E$ for $i \in 1, \ldots, n-1$. We denote P as a v_1–v_n path. If $v_1 = v_n$, then P is a cycle.*

If G is weighted, the weight of P is the summed weight of the edges on P, denoted by $c(P) = \sum_{i=1}^{n-1} c((v_i, v_{i+1}))$.

Note that in the case of $c : E \to \mathcal{F}(\mathbb{R}, \mathbb{R})$ we define the "sum" of two functions as a special *link* operation, see Definition 2.16 on page 10.

Definition 2.9. *DAG*

A directed acyclic graph*, or DAG, is a directed graph with no cycles.*

Definition 2.10. *Tree*

A tree *is a DAG where each pair of vertices is at most connected by a single path.*

Definition 2.11. *Spanning Tree*

A spanning tree *of a graph G is a subgraph of G which contains all vertices of G and is a tree.*

Definition 2.12. *Shortest Path Problem*

Given a weighted graph G with cost function $c : E \to \mathbb{R}_+$, a source vertex s and a target vertex t, find the path from s to t with minimal weight.

The Shortest Path Problem can be solved in polynomial time [Dij59]. In a graph with a real cost function, we denote the weight of a shortest s–t path by $\mathrm{d}(s,t)$: the *shortest path distance* from s to t, or simply the distance from s to t.

Definition 2.13. *Shortest Path Tree*

A shortest path tree T *is a spanning tree of $G = (V, E)$ rooted at $v \in V$, such that the path distance from v to any vertex u in T is the shortest path distance from v to u in G.*

Dijkstra's Algorithm. When finding a shortest path from s to t in a graph G with a cost function $c : E \to \mathbb{R}_+$, Dijkstra's algorithm is the standard basis for most approaches. As seen in Algorithm 2.1 on the next page, the vertices in G are visited in the order of their distance from s. After v is extracted from the priority queue, the distance from s to v is known.

An improvement of Dijkstra's algorithm is using a *stopping criterion*. As soon as t is extracted from the queue, the algorithm can terminate: the distance from s to t is known. Furthermore, the algorithm is *vertex setting*. Each vertex is guaranteed to be extracted only once from the queue. Given time complexities of the queue operations T_{insert}, $T_{\text{deleteMin}}$ and $T_{\text{decreaseKey}}$, the running time of the algorithm is in $O(n \cdot (T_{\text{insert}} + T_{\text{deleteMin}}) + m \cdot T_{\text{decreaseKey}})$.

Several additional annotations are associated with Dijkstra's algorithm. The procedure in lines **9** to **15** is called *edge relaxation*. We check whether an edge (u, v) offers a shorter distance to v via u, and if so store the new distance at v. Whenever we relax $e = (u, v)$ we say that v is *visited*. Whenever we extract v from the priority queue, we say that v is *settled*.

A *bidirectional* Dijkstra s–t query in G consists of a *forward* and a *backward* search, both done by Dijkstra's algorithm. In the forward search we simply start Dijkstra from s. The backward search runs from t and works on the reverse graph G_{r}. At each vertex v we store a forward distance $\mathrm{d}(s, v)$ computed by the forward search, and a backward distance $\mathrm{d}(v, t)$ computed by the backward search. We then define the *tentative distance* to t as $\min_{v \in V} \mathrm{d}(s, v) + \mathrm{d}(v, t)$: the best known distance to t. Note that before the two searches meet at a vertex, the tentative distance is ∞.

Path Extraction. After computing the distance from s to t with Dijkstra's algorithm, we may extract a shortest s–t path. An edge $e = (u, v)$ relaxed by Dijkstra lies on a shortest path, iff $\mathrm{d}(s, u) + c(e) = \mathrm{d}(s, v)$ after the algorithm terminates. In order to reach the target vertex, the algorithm relaxed a path of edges to t. Thus, we may start from t and follow ingoing edges that lie on shortest paths until we reach s, in order to construct a shortest s–t path.

Definition 2.14. *Constrained Shortest Path Problem*

Given a weighted graph G with edge cost function $c : E \to \mathbb{R}_+$, a resource function $r : E \to \mathbb{R}_+$, a real value $R \in \mathbb{R}_+$, a source vertex s and a target vertex t, find a path P from s to t, s.t. $c(P)$ is minimal and $\sum_{e \in P} r(e) \leq R$.

The Constrained Shortest Path Problem is NP-complete [Jaf84].

Algorithm 2.1: DIJKSTRA

Input: Graph $G = (V, E, c)$, source vertex s
Data: Priority queue Q
Output: Distances $\mathsf{d}(v)$ for all $v \in V$, shortest-path tree of s given by $\mathsf{pred}(\cdot)$

```
// Initialization
1  forall v ∈ V do
2      d(v) ← ∞
3      pred(v) ← null
4  d(s) ← 0
5  Q.INSERT(s, d(s))
// Main loop
6  while Q is not empty do
       // Extract unsettled vertex
7      u ← Q.DELETEMIN()
       // Iterate outgoing edges
8      forall (u, v) ∈ E do
9          if d(u) + c((u, v)) < d(v) then
10             d(v) ← d(u) + c((u, v))
11             pred(v) ← u
12             if Q.CONTAINS(v) then
13                 Q.DECREASEKEY(v, d(v))
14             else
15                 Q.INSERT(v, d(v))
```

Definition 2.15. *Multi-objective Shortest Path Problem*

Given a weighted graph G with cost function $c : E \to \mathbb{R}^k$, a source vertex s and a target vertex t, find the set of paths $\mathcal{P}$ from s to t, s.t. $\{c(P) \mid P \in \mathcal{P}\}$ is a maximal Pareto set.

The Multi-objective Shortest Path Problem is NP-complete [MD79].

2.3. Time-Dependent Routing

In standard vehicle routing we wish to know the earliest arrival time from s to t in a street network. To compute the arrival time we first model the network as a weighted directed graph, where edges represent streets segments and vertices model intersections or street bends. The cost function $c : E \to \mathbb{R}$ corresponds to the travel time we require to traverse each edge.

In time-dependent routing we also wish to compute earliest arrival times, however we take traffic situations into account. Instead of having static edge costs, i.e. one real value per edge, we have a cost function $c : E \to \mathcal{F}(\mathbb{R}, \mathbb{R})$. Thus, with each edge e we associate an *edge cost function* $c(e)$, which represents the travel time we require to traverse the edge at different points in time (e.g. times of a day). Figure 2.3 on the following page shows an example of a typical piecewise linear function used in time-dependent routing. During the rush hours of a day the edge cost rises, and in the less populated hours it falls.

We can then use the edge functions as follows. Given a source vertex s and a starting point in time τ, compute the earliest arrival at a target vertex t. This query can be answered by applying Dijkstra's algorithm, as we have the specific point in time at which we arrive at

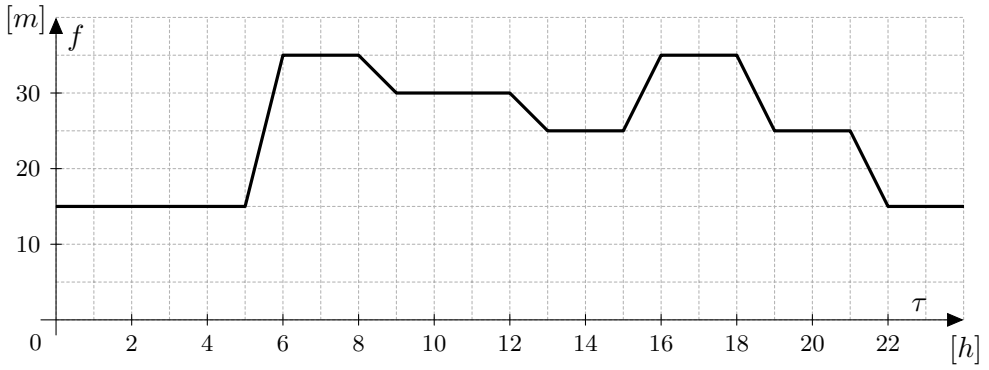


Figure 2.3.: An example of an edge cost function used in time-dependent routing. Midnight and midday hours have relatively low edge costs, whereas morning and evening hours have an increased edge cost due to traffic congestions.

each vertex v. We can therefore evaluate the edge function of $e = (v, u)$, which yields a standard non-negative weight. The result of this query is a shortest s–t path w.r.t. τ.

A more interesting query is the *profile search*. Given a source vertex s, compute the arrival times at a target vertex t, dependent on a parameter τ. Obviously, the result is a function which depends on the edge cost functions. To answer a profile search, we first observe two major differences between standard routing and time-dependent routing.

First, in the standard scenario traversing consecutive edges $e_1 = (u, v)$ and $e_2 = (v, w)$ results in the summed travel times $c(e_1) + c(e_2)$. In the time-dependent case, we arrive at u with a certain travel time τ. Traversing e_1 requires $c(e_1)(\tau)$ travel time, and so we arrive at v at $\tau + c(e_1)(\tau)$. Thus, we require $c(e_2)(\tau + c(e_1)(\tau))$ travel time to traverse e_1 and e_2. We define this composition as the time-dependent *link* operation.

Definition 2.16. *Link (time-dependent)*

Given edge cost functions f and g, the link result in the time-dependent case is defined as

$$(f \circ g)(\tau) := g(\tau + f(\tau))$$

In the case of piecewise linear functions, the link operation complexity lies in $\mathcal{O}(n + m)$, where n and m are the number of interpolation points of f resp. g [DW09]. Note that after defining the link operation we may now compute the path cost function $c(P)$ of a path P. That is, $c(P) := c(e_1) \circ c(e_2) \circ \ldots \circ c(e_n)$, where $P = \{v_1, \ldots, v_{n+1}\}$ and $e_i = (v_i, v_{i+1}), i \in 1, \ldots, n$.

And second, when arriving at a vertex v in the standard case, via different paths P_1 and P_2, we are only interested in the minimum travel time to v. We can therefore discard the path with greater travel time as it is dominated. In the time-dependent scenario we may have $c(P_1) \not\propto c(P_2)$, as seen on Figure 2.4 on the next page. Thus, to achieve minimal travel time, we use the minimum of $c(P_1)$ and $c(P_2)$. This defines the *merge* operation, which allows us to choose the best path to v for a given τ.

Definition 2.17. *Merge (time-dependent)*

Given edge cost functions f and g, the merge result in the time-dependent case is defined as

$$(f \cup g)(\tau) := \min(f(\tau), g(\tau))$$

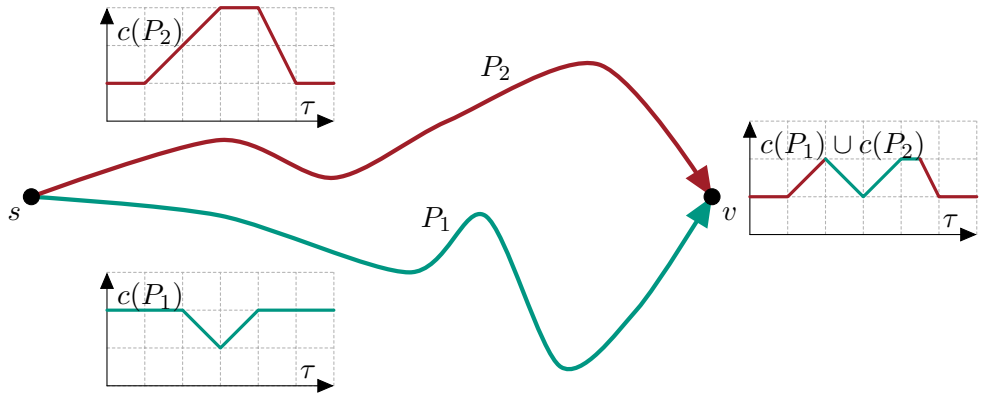


Figure 2.4.: Paths P_1 and P_2 from s to t, with $c(P_1) \not\preccurlyeq c(P_2)$. Departing at point in time τ, to arrive at v with minimal travel time we use the minimum of $c(P_1)(\tau)$ and $c(P_2)(\tau)$. The resulting function at v is then $c(P_1) \cup c(P_2)$.

Merging two piecewise linear functions f and g has a linear complexity in the size of $f \cup g$ [DW09]. Having defined the link and merge operations in the time-dependent case, we examine Dijkstra's variant used to compute profiles.

Profile Dijkstra. In the case of a cost function $c : E \to \mathcal{F}(\mathbb{R}, \mathbb{R})$, we can alter Dijkstra's algorithm to answer profile queries. Instead of propagating distances as in the original algorithm, we propagate piecewise linear functions. See Algorithm 2.2 on the following page. We refer to the resulting function at t (after an s–t query) as the *s–t profile*.

The starting function at s is set to an initial value, for instance $0 \in \mathcal{F}(\mathbb{R}, \mathbb{R})$. We then use the link operation whenever we traverse an edge $e = (u, v)$. That is, we take the current function at u, $f(u)$, and compute $f(u) \circ c(e)$. We then check whether a merge of $f(v)$ with $f(u) \circ c(e)$ is required. If so, we store the merge result $f(v) \cup (f(u) \circ c(e))$ at v.

The above operation replaces Dijkstra's edge relaxation. However, if $f(u) \circ c(e)$ is merged with $f(v)$ we must propagate the improved portion of the result. Thus, we must insert v in the priority queue, even if we already extracted v previously. Thus, we no longer settle each vertex once, as in Dijkstra's algorithm: the function at each vertex may be corrected multiple times. The altered algorithm is *label correcting*.

For the queue order we may use any type of ordering imposed on real functions, e.g. $f(v)_{\text{key}} := \min_{x \in \mathbb{R}} f(v)(x)$. Here $f(v)(x)$ is the current function stored at v, evaluated at x.

Due to the merge operation, the functions we propagate become increasingly complex and so require an increasing number of interpolation points. In large graphs profile searches are prohibitive, as the link and merge operations become costly with the progression of the search [DW09].

2.4. Electric Vehicle Routing

Electric vehicle routing (or simply EV routing) is distinguished from standard routing due to the vehicles' different power supply. Compared to standard vehicles, the cruising range of electric vehicles is much more limited and recharging stations are not as widely spread as gas stations. Furthermore, unlike traditional vehicles, electric vehicles may recuperate energy when braking or traveling downhill. To better integrate the battery constraints into a routing algorithm, several important notions were introduced by Eisner et al. [EFS11] and Baum et al. [BDPW13]:

Algorithm 2.2: PROFILE DIJKSTRA

Input: Graph $G = (V, E, c)$, source vertex s, target vertex t
Data: Priority queue Q
Output: Optimal function $\mathrm{F}(t)$ to t w.r.t. c.

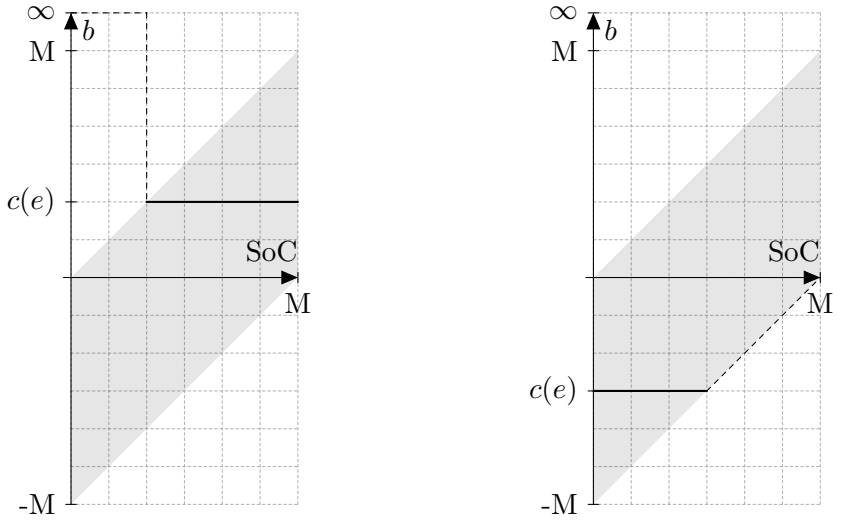
```
// Initialization
forall v ∈ V do
    F(v) ← ∞
F(s) ← 0 : ℝ → 0
Q.INSERT(s, F(s).KEY())
// Main loop
while Q is not empty do
    // Extract unsettled vertex
    u ← Q.DELETEMIN()
    // Iterate outgoing edges
    forall (u, v) ∈ E do
        f' ← F(u).LINK(c((u, v)))
        if F(v) ⋠ f' then
            F(v).MERGE(f')
            if Q.CONTAINS(v) then
                Q.DECREASEKEY(v, F(v).KEY())
            else
                Q.INSERT(v, F(v).KEY())
```

- Battery Capacity. The electric vehicle battery's capacity is denoted by M, typically in [mWh].
- SoC. Denotes the current energy state of the battery, i.e. the state of charge. The value range of SoC lies within $[0, \mathrm{M}]$.
- Consumption. Defined as $\mathrm{C} = \mathrm{M} - \mathrm{SoC}$, consumption yields the difference between the energy of a fully charged battery and the current charge of the battery. Due to $\mathrm{SoC} \in [0, \mathrm{M}]$, C may not drop below 0 or exceed M.
- Undercharging. If the current SoC drops below 0 undercharging occurs. The electric vehicle's battery has no energy left and so the vehicle cannot be driven further.
- Overcharging. Whenever the electric vehicle brakes or is driven downhill, it is possible that the recuperated energy, added to the current SoC, exceeds the battery capacity M. As the battery cannot be charged higher than its capacity, the recuperated amount is cut at $\mathrm{M} - \mathrm{SoC}$.

To handle undercharging and overcharging, Eisner et al. further define the energy consumption function $\mathrm{b} : E \to \mathcal{F}(\mathbb{R}, \mathbb{R})$. Given an edge e with energy cost $c(e)$, b yields a function which maps input SoC to consumption cost required to traverse e:

$$\mathrm{b}(e)(\mathrm{SoC}) = \begin{cases} \infty & \mathrm{SoC} - c(e) < 0, \\ \mathrm{SoC} - \mathrm{M} & \mathrm{SoC} - c(e) > \mathrm{M}, \\ c(e) & \text{otherwise} \end{cases}$$

In the case of undercharging, the energy cost of the function is set to infinity, as the electric vehicle cannot traverse the edge. If overcharging occurs, b ensures that applying $\mathrm{b}(e)$ to



(a) An edge with a positive cost. (b) A downhill edge with a negative cost.

Figure 2.5.: An example of the function b. The grey area illustrates the allowed consumption costs, depending on the current SoC. States of charge that are too low are mapped to ∞, as undercharging occurs. Negative energy cost that is too low, causing overcharging, is clipped to the grey area so that $\mathrm{SoC} - \mathrm{b}(e)(\mathrm{SoC})$ does not exceed M.

the current SoC obeys the battery capacity. Figure 2.5 shows an example of b at both an edge with a positive cost and a downhill edge with a negative cost.

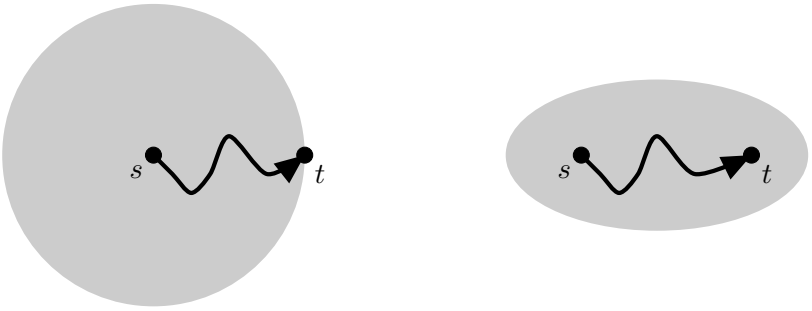
As in time-dependent routing, we may use b as a cost function to compute a shortest path from s to t. However, as the cost function yields energy cost, we compute the most energy efficient path from s to t. If we want to compute a shortest path in terms of travel time, we need to solve the multi-objective or constrained shortest path problem.

Either approach is hindered by a further problem. As seen in Figure 2.5b, energy recuperation results in negative consumption. Dijkstra's algorithm, however, only allows non-negative edge weights. As negative cycles are physically impossible, we may use the Bellman-Ford algorithm [Bel56] which runs in $\mathcal{O}(n \cdot m)$. To avoid the asymptotically worse running time, however, we can apply vertex potential shifting and still use Dijkstra's algorithm.

Potential Shifting (Johnson's Algorithm). Let G be a weighted graph with an edge cost function $c : E \to \mathbb{R}_+$, where some edges have negative weights but no negative cycles exist. Johnson's algorithm [Joh73] introduces *vertex potentials* defined by the function $\pi : V \to \mathbb{R}$. The distance $\mathrm{d}(s, v)$ is then redefined as $\mathrm{d}(s, v) + \pi(v)$, effectively changing the order in which vertices are visited by Dijkstra. As Dijkstra's algorithm uses edge weights, and not vertex weights, the *reduced edge costs* are defined. The reduced cost of $e = (u, v)$ is set to $c(e)' = \pi(v) + c(e) - \pi(u)$. The reduced cost of a path $P = (v_1, \ldots, v_n)$ is then $c(P)' = \pi(v_n) + c(P) - \pi(v_1)$.

A vertex potential is feasible, iff for all $e \in E$ the reduced edge cost $c(e)'$ is non-negative [Joh73]. For this reason, a feasible vertex potential is of particular interest in EV routing. With the help of the potential, there are no negative reduced edge costs and so we may run Dijkstra to compute the most energy efficient s–t path. To compute the shortest travel time path, we may now use the multi-criteria variant of Dijkstra [Sto12].

Height-induced Potentials. A height-based method of computing feasible vertex potentials is offered by Baum et al., [BDPW13]. The potential of a vertex $v \in V$ is defined as



(a) Dijkstra's algorithm. (b) The A* algorithm.

Figure 2.6.: Search space shapes of Dijkstra's and the A* algorithms. Dijkstra's search space is a graph theoretical circle with radius $\mathrm{d}(s,t)$. The search space of A*, when applying a meaningful potential π, is an ellipse growing in the direction of t. The goal direction results in fewer settled vertices before extracting t from the priority queue.

$\pi(v) := \gamma \cdot h(v)$, where $h(v)$ is the altitude of the vertex. The edges in E are split in *downward* and *upward* edges, $E^{\downarrow} := \{(u,v) \in E \mid h(u) > h(v)\}$ resp. $E^{\uparrow} := \{(u,v) \in E \mid h(u) \leq h(v)\}$. Then they define $\underline{\gamma}$ and $\overline{\gamma}$ as:

$$\underline{\gamma} = \max_{(u,v)\in E^{\downarrow}} \frac{c((u,v))}{h(v)-h(u)}, \overline{\gamma} = \min_{(u,v)\in E^{\uparrow}} \frac{c((u,v))}{h(v)-h(u)}$$

In case that $\underline{\gamma} \leq \overline{\gamma}$, setting $\gamma := \underline{\gamma}$ yields a feasible potential. Furthermore, a single linear sweep over all edges yields potentials usable in any query.

A* Search. The A* approach is a goal directed speed-up technique for Dijkstra's algorithm [HNR68a]. Starting from the source node $s \in G$, the search space of Dijkstra's algorithm is an expanding (graph-theoretic) circle with radius equal to the current minimum key in the priority queue. The stopping criterion suggested earlier ensures that we stop expanding this circle as soon as it meets the target vertex t, as seen in Figure 2.6a. The search is not goal directed, as it expands in all directions with equal pace. The "area" of the search space can be reduced by changing the search space's shape to an ellipse growing towards t, see Figure 2.6b. To achieve this different shape, vertex potentials are used as in Johnson's algorithm.

To direct the search space towards the target vertex t, the cost of edges leading to t is lowered. Setting $\pi(u) = \mathrm{d}(u,t)$ results in $c(e)' = \mathrm{d}(v,t) + c(e) - \mathrm{d}(u,t)$ for $e = (u,v) \in E$. If e lies on a shortest path to t, then the equality $\mathrm{d}(u,t) = c(e) + \mathrm{d}(v,t)$ yields reduced edge cost $c(e)' = 0$. As Dijkstra's algorithm requires non-negative edge costs, this is the optimal vertex potential: the search will first visit edges on shortest paths to t. Of course, $\mathrm{d}(u,t)$ is not known for $u \in V$ and so has to be approximated. Here only lower bounds on $\mathrm{d}(u,t)$ may be used: otherwise the vertex potential is no longer feasible [GH05]. The smaller the difference between lower bounds and actual distances to t is, the better the search is guided to t.

Multi-criteria Dijkstra. To solve the multi-criteria shortest path problem, again with non-negative edge weights, we can alter Dijkstra's algorithm. The outline of the algorithm can be seen in Algorithm 2.3 on the next page. Instead of storing a single distance value at each vertex v, we store a (tentative) Pareto set $\mathcal{B}(v)$. The Pareto set $\mathcal{B}(v)$ represents the distance from s to v. We say that the elements of $\mathcal{B}(v)$ are the *labels* at v.

Algorithm 2.3: MLC Dijkstra

Input: Graph $G = (V, E, c)$, source vertex s, target vertex t
Data: Priority queue Q
Output: Pareto-set $\mathcal{B}(t)$ w.r.t. c.

```
   // Initialization
 1 forall v ∈ V do
 2     B(v) ← ∅
 3 B(s) ← 0 ∈ ℝ^k
 4 Q.insert(s, B(s).key())
   // Main loop
 5 while Q is not empty do
       // Extract unsettled element in bag
 6     u ← Q.peekMin()
 7     e ← B(u).nextUnsettled()
 8     if ¬B(u).hasUnsettled() then
 9         Q.deleteMin ()
       // Iterate outgoing edges
10     forall (u, v) ∈ E do
11         e' ← e + c((u, v))
12         if B(v) ⋠ e' then
13             B(v) ← { p ∈ B(v) | e' ⋠ p } ∪ {e'}
14             if Q.contains(v) then
15                 Q.decreaseKey(v, B(v).key())
16             else
17                 Q.insert(v, B(v).key())
```

Similar to the profile variant of Dijkstra, the multi-criteria algorithm may visit the same node multiple times. However, each label $\ell \in \mathcal{B}(v)$ is propagated only once by the algorithm. The algorithm is *label setting*.

Due to the loss of the vertex setting property, t can be extracted multiple times from the queue and the stopping criterion cannot be used. We may, however, use *target pruning*. Whenever a vertex v is extracted from the queue and the next unsettled label $\ell \in \mathcal{B}(v)$ is chosen, we check whether $\mathcal{B}(t) \propto \ell$. If so, we continue with the next minimal element in the queue.

And last, we require keys for the priority queue. For this, we set the key of $\ell \in \mathcal{B}(v)$ as a linear combination of ℓ's components: $\ell_{\text{key}} := \alpha \cdot \ell, \sum \alpha_i = 1$.

2.5. Contraction Hierarchies

Contraction Hierarchies (CH) [GSSD08] are an adaptation of Dijkstra's algorithm to street networks. As already seen, Dijkstra's algorithm settles vertices based only on graph structure and edge weight information. The CH algorithm, on the other hand, utilizes the hierarchical structure of street networks. Traveling between two cities provides an example of this structure. Intuitively, local city streets are only of importance until we reach a highway connecting the two cities, or after we leave the highway in the destination city. Even more, we are not interested in local streets of other cities. Contraction Hierarchies apply this observation to skip over large (unimportant) portions of the street network graph during the shortest path search.

To achieve this, CH separates the shortest path query in two stages. The first stage is an offline preprocessing step, which is done once and is computation heavy. During this preprocessing parts of the graph are contracted and replaced by shortcuts, retaining shortest path information.

The second stage is where the actual query takes place. Dijkstra's algorithm is run on the contracted graph, using shortcuts in place of regular edges. In this manner the unimportant (for the query) parts of the graphs are skipped by the algorithm, and the search space is reduced.

The standard CH algorithm computes shortest paths in a graph G with positive real weights.

Preprocessing. The offline part of CH has two components. First, the *vertex contraction* procedure. When contracting a vertex $v \in G$, v is "removed" from G. To preserve the shortest path informations in G, each pair (u, w) of v's neighbors is examined. If v lies on the only shortest u–w path, then a shortcut from u to w with weight $\mathrm{d}(u, w)$ is added, and v is removed from G. Determining whether v lies on the only shortest u–w path is commonly named a *witness search*. The witness search is usually done in a limited search space, and failing to find a shortest u–w path (which does not pass through v) results into a u–w shortcut. This conservative approach retains the correctness of the algorithm: the witness search was stopped before knowing whether the only shortest u–w path contains v.

As a shortest path query with $s = v$ is possible, v is not actually removed from G upon contraction. Instead, v is "pushed down" by setting v's *level* to be one less than the level of v's neighbours during the contraction. Further, when contracting v any already contracted neighbour is ignored. This results in a layered graph based on the vertex levels, where the vertices in higher layers (levels) are considered to be more important then vertices in lower layers.

The second component of the preprocessing is the *vertex ordering*, which corresponds to the vertex importance in the street network. Starting from the least important vertex according to the ordering, vertices are contracted one by one. This is done until all vertices are contracted.

The quality of the so created contraction hierarchy depends mainly on the vertex ordering. Thus, multiple publications exist on the topic of a qualitative vertex order: [Gei08, Mil12, DSW14].

Query. The s–t CH query consists of a bidirectional Dijkstra s–t query on the contracted graph. Both searches treat shortcuts as regular edges, and may use only edges (and shortcuts) leading to vertices in higher levels. As in the bidirectional variant of Dijkstra's algorithm, both a forward and a backward distance to each vertex is stored.

As soon as both searches terminate, the shortest path from s to t is found. It is composed of two shortest paths: from s to v and from v to t. Here, v is the vertex with least summed forward and backward distances. The proof of correctness of this approach is found in [GSSD08]. Due to properties of the contraction operation and the order of contractions, CH ensures that the search spaces of the two directions have a common vertex which lies on a shortest s–t path.

The standard stopping criterion used in the bidirectional Dijkstra algorithm is to terminate both searches, when the sum of the priority queues' minimum keys exceeds the tentative distance. During the CH query, this criterion is no longer correct [Gei08], as both searches are prohibited from relaxing edges leading to vertices in lower levels. A more conservative stopping criterion, which ensures correctness [Gei08], functions as follows. A search direction

terminates as soon as its queue minimum key exceeds the tentative distance to any vertex common for both directions.

Partial Contraction Hierarchies. Some techniques, such as CALT [BDS+08], only partially contract the input graph leaving a small percentage of vertices (0% to 1%) at the top of the hierarchy. The set of uncontracted vertices is referred to as the *core* of the CH.

2.6. Time-Dependent Contraction Hierarchies

In time-dependent routing the standard CH algorithm is not applicable, as the edge costs are functions. More specifically, during the standard time-dependent s–t query the arrival time at a settled vertex is known. It is therefore possible to evaluate the function at each edge e during the relaxation of e. Computing the earliest arrival at t is thus possible with Dijkstra's algorithm. During the CH preprocessing, however, the arrival time at vertices is not known. The preprocessing must ensure that the CH query is correct for each source and target vertices, at any starting point of time τ [BGNS10].

The natural approach to solve this problem is to utilize the profile search. As a single weight on each shortcut no longer suffices, functions computed by the profile variant of Dijkstra are stored. A shortcut (u, w) between the neighbours u and w of v is not required, if removing v from G does not change the u–w profile [BGNS10]. In other words, for all starting points of time τ there exists a shortest path from u to w that does not pass through v.

The query step of CH also requires adaptation if profiles are stored at the shortcuts. No change is required in the forward search, as the arrival time is known at the visited vertices and so each edge cost function can be evaluated. The backward search, on the other hand, has no information on the arrival time. Thus, a backward search is ran which computes a lower and an upper bound, instead of whole profiles. The forward search then continues into the backward search space. For this second phase of the forward search, only edges which lead to lower levels are used. The s–t distance is known as soon as t is extracted from the priority queue of the forward search.

To answer a profile query, a *corridor search* is used [BGNS10]. In this approach each shortcut stores a lower and an upper bound of the actual profile which the shortcut normally represents. The bounds have a limited number of interpolation points, reducing the cost of the merge and link operations. The actual CH query first computes a corridor of edges in which the solution profile lies, and then proceeds to compute the exact profile only in this corridor.

Function Simplification. In [BGNS10] Batz et al. apply the Imai-Iri piecewise linear function simplification, [II87]. This algorithm supports only continuous functions, which in the time-dependent setting is not an issue. However, the profile functions we introduce in Chapter 3 allow discontinuities. We may thus not apply the same function simplification without limiting the algorithm to continuous portions of the functions. The effectiveness of the Imai-Iri algorithm is thus hindered by the number of discontinuous points of a function.

Instead, we resort to the greedy function simplification of Visvalingam and Whyatt [VW93]. Let f be a piecewise linear function with interpolation points $\{p_1, \ldots, p_n\}$. In the following we say that f has a *convex / concave bend* at three consecutive points $\{p_{i-1}, p_i, p_{i+1}\}$, iff the slope of $\overline{p_{i-1}p_i}$ is less / greater than that of $\overline{p_ip_{i+1}}$. Similarly, f has a *convex / concave turn* at four consecutive points $\{p_{j-1}, p_j, p_{j+1}, p_{j+2}\}$, iff the slopes of $\overline{p_{j-1}p_j}$, $\overline{p_jp_{j+1}}$ and $\overline{p_{j+1}p_{j+2}}$ are monotonically increasing / decreasing.

Suppose we wish to simplify f by obtaining a function $\overline{f}$ where $f \propto \overline{f}$. The simplification routine examines all convex bends and all concave turns of f. The middle point p_i of

a convex bend $\{p_{i-1}, p_i, p_{i+1}\}$ may be removed from f, resulting in a function which is dominated by f. Here, the introduced maximum error is equal to the distance from p_i to the segment $\overline{p_{i-1}p_{i+1}}$. In the case of a concave turn $\{p_{j-1}, p_j, p_{j+1}, p_{j+2}\}$, the middle points p_j and p_{j+1} may be replaced by the intersection p' of the lines through $\overline{p_{j-1}p_j}$ and $\overline{p_{j+1}p_{j+2}}$. Again, the result is a function dominated by f. The maximum error is then the distance from p' to $\overline{p_j p_{j+1}}$.

The simplification routine repeatedly removes or replaces interpolation points from f, each time choosing the convex bend or concave turn where the maximum introduced error is minimized. Note that after each such removal or a replace, new bends or turns may result. In addition, to avoid quadratic complexity a priority queue is used to determine the bend or turn with smallest error.

The same approach may be applied to obtain a simplified lower bound of f. To do so, the routine consecutively reduces the concave bends and the convex turns of f.

3. Profile Operations

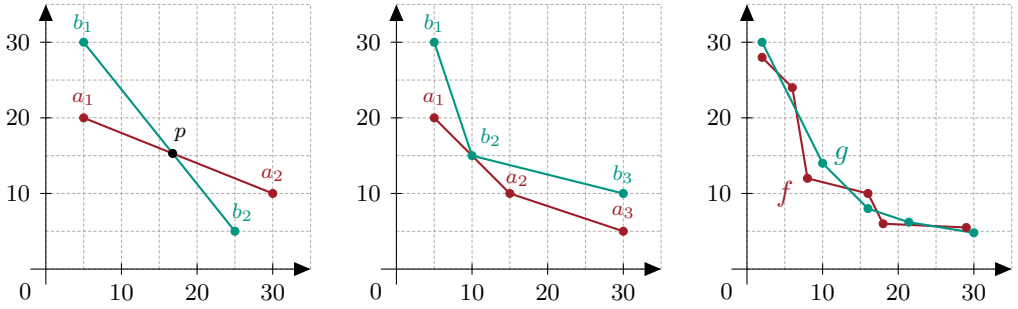
In electric vehicle routing traveling at different speeds has an impact on energy consumption and thus on the cruising range. We are therefore interested in a mapping from travel time to consumption, given a source vertex s and a target vertex t in the street network graph. The mapping should indicate the minimal energy consumption C, if we choose to invest travel time τ in a path from s to t.

To approximate such a mapping, Baum et al. [BDHS$^+$14] model the street network as a *multigraph* $G = (V, E)$. A multigraph allows multiple (u, v) edges, where $u, v \in V$. Between two adjacent vertices u and v multiple edges exist, representing the variable traveling speed on the street segment from u to v. The weight of each edge $e = (u, v) \in E$ is a travel time and energy consumption *tradeoff point* $(\tau, \mathrm{C}) \in \mathbb{R}^2$. Baum et al. then run a multi-criteria s–t Dijkstra query. The resulting Pareto set at t is the approximation of the desired mapping.

Using this approach we are only allowed to traverse a street segment with a fixed number of travel times. In reality, we may also traverse the segment with any of the in-between travel times. We achieve this by interpreting the tradeoff points between the adjacent vertices u and v as the interpolation points of a piecewise linear function. Between two tradeoff points p_1 and p_2 the linear interpolation $\lambda \cdot p_1 + (1 - \lambda) \cdot p_2, \lambda \in [0, 1]$, results from traveling at the speed of p_1 for the first λ-portion of the segment, and the rest of the segment at the speed of p_2. We may then use a profile search (as in time-dependent routing) to compute the desired mapping from travel time to energy consumption.

As the parallel edges with different costs are now represented by a single edge with a cost function, we no longer require a multigraph. Thus, we use a weighted graph G with a cost tradeoff function $c : E \to \mathcal{F}(\mathbb{R}, \mathbb{R})$. We denote the edge cost function $c(e)$ for $e \in E$ as a *tradeoff function*. The process of generating the tradeoff points of edges we adopt directly from Baum et al.

Tradeoff Functions. The faster an electric vehicle travels, the higher the energy consumption is. The interpolation points of a tradeoff function therefore have monotonically increasing travel time values, and monotonically decreasing energy consumption values. In short, our tradeoff functions are *monotonic*. Owing to aerodynamic drag [LL04], the force required to propel an electric vehicle is quadratic in the driving speed. Thus, on a street segment, the function which maps travel time to consumption is a convex curve: higher speeds reduce travel time and increase (quadratically) consumption. For this reason the tradeoff edge cost functions are also convex.



(a) Inner intersection. (b) End-point intersection. (c) Multiple intersections.

Figure 3.1.: Examples of an inner intersection, an intersection at an end point and two functions intersecting at $\mathcal{O}(n+m)$ points.

The leftmost interpolation point of a tradeoff function $c(e)$ defines the minimum travel time $\min_\tau$ and maximum energy consumption $\max_C$ required to traverse e. The rightmost interpolation point likewise defines the maximum travel time $\max_\tau$ and minimum energy consumption $\min_C$. Note that for $\tau < \min_\tau$ our definition of a piecewise linear function sets the energy consumption to ∞. Similarly, for $\tau > \max_\tau$ the consumption remains constant and equal to $\min_C$: we gain no further benefit by traveling at speeds slower than the minimum speed set for e.

Having chosen the graph setting, we now need to define the merge and link operations for the tradeoff functions. Furthermore, we require algorithms for the computation of linking and merging, as well as testing whether a piecewise linear function dominates another. We can then proceed with a standard profile search algorithm and build upon it.

3.1. Merge Operation

The definition of the merge operation in the electric vehicle routing case does not differ from the time-dependent variant, see Definition 2.17 on page 10. To compute the merge of two tradeoff functions f and g we use Algorithm 3.1 on the next page.

As seen in the pseudocode of the routine, we first introduce (virtual) auxiliary points to both functions. These help representing the vertical and horizontal segments with which the tradeoff functions begin resp. end, exposing all segments of f and g for the algorithm. We then check whether $a_1 \propto b_1$, i.e. f allows less minimum travel time $\min_\tau$ than g, or equal $\min_\tau$ and less maximum energy consumption $\max_C$. In this case f overlaps with the merge result $m := f \cup g$ until the first segment intersection of f and g. Otherwise, g overlaps with m in this manner. If no segment intersection exists, then one function dominates the other and m overlaps entirely with the dominating function. We now examine the other case, i.e. an intersection exists.

Let p be the first segment intersection of f and g w.r.t. the travel time axis, i.e. p is the leftmost intersection. Let $p \in c$ and $p \in d$, where $c := \overline{a_i a_{i+1}}$ resp. $d := \overline{b_j b_{j+1}}$. Further assume p is an inner intersection, i.e. p does not coincide with a_i, a_{i+1}, b_j or b_{j+1}. The minimum m must therefore "switch" from one function to the other at p, as the slopes of c and d differ in order to yield the intersection p (p is inner and first). E.g. if c offers less energy consumption than d before p, then d must have a lesser slope than c for p to occur and so d allows less energy consumption than c after p (see Figure 3.1a). It is therefore enough to keep track of the current minimal function and switch to the other function at an inner intersection.

We may apply the same observation for arbitrary intersections, however in some cases an intersection which is not inner does not call for a switch. For instance, if $p = b_{j+1}$ and c

Algorithm 3.1: MERGE

Input: Tradeoff functions f and g defined by interpolation points $a_1, \ldots, a_n$ resp. $b_1, \ldots, b_m$

Output: Merged tradeoff function $m := f \cup g$

```
   // Explicit first and last segments of f and g
1  a_0 ← (f.min_τ, ∞), a_{n+1} ← (∞, f.min_C)
2  b_0 ← (g.min_τ, ∞), b_{m+1} ← (∞, g.min_C)
3  l_a ← f.min_τ < g.min_τ or (f.min_τ = g.min_τ and f.max_C ≤ g.max_C)
4  m ← ∅
5  i ← 0, j ← 0
   // Sweep functions
6  while i ≤ n and j ≤ m do
       // Add current point of lower function to result
7      if l_a then m.APPEND(a_i)
8      else m.APPEND(b_j)
       // Intersection test, lower function may become upper if positive
9      c ← a_i a_{i+1}, d ← b_j b_{j+1}
10     if c intersects d then
11         p ← INTERSECTION(c,d)
12         t ← p.τ + ε
           // Evaluate f and g at p.τ + ε, using segments on which p lies
13         if l_a ⇒ f(t) > g(t) and ¬l_a ⇒ f(t) < g(t) then
14             m.APPEND(p)
15             l_a ← ¬l_a
       // Move to next segment of f or g, whichever starts "earlier"
16     if a_{i+1}.τ < b_{j+1}.τ then i ← i + 1
17     else j ← j + 1
```

offers less energy consumption than both d and $\overline{b_{j+1}b_{j+2}}$ then m continues to overlap with f after p (Figure 3.1b). Testing whether this case occurs is simple: we compare f and g at $t := \tau + \epsilon$, where τ is the travel time of p. If the current minimal function continues to be less than the other function at t, then no switch is required. Note that during lines **10** to **15** of Algorithm 3.1, evaluating f and g at t is possible in constant time – as we are already at the segments containing p, we must either interpolate those segments, or the directly sequential segments of f resp. g.

Algorithm 3.1 starts with the minimal function at $\min\{\min_\tau^f, \min_\tau^g\}$, where $\min_\tau^f$ and $\min_\tau^g$ are the minimum travel times of f resp. g. The algorithm then chooses the minimal function (f or g) between consecutive intersections, lines **12** to **15**. Correctness follows from the fact that the algorithm inspects all intersections between f and g.

Lemma 3.1. *Given tradeoff functions f and g with n resp. m interpolation points, $f \cup g$ has $\mathcal{O}(n+m)$ interpolation points.*

Proof. Throughout the proof we use the fact that f and g are monotonic ($*$).

Let f and g be defined by interpolation points $a_1, \ldots, a_n$ resp. $b_1, \ldots, b_m$. Consider a segment $s = \overline{a_i a_{i+1}}$ of f where $a_i = (\tau_1, \mathrm{C}_1)$ and $a_{i+1} = (\tau_2, \mathrm{C}_2)$. Either g has some interpolation points $b_j, \ldots, b_{j+k}$ with travel time in the interval $I := [\tau_1, \tau_2]$ or none. In

the latter case g may intersect s at most once due to $(*)$: g has at most 1 segment which allows travel times within I. In the former case, s may intersect g at most $k \cdot 2$ times in I – a maximum of 2 intersections for each two segments adjacent to a point in $b_j, \ldots, b_{j+k}$. Again, we use $(*)$: segments of g share travel times at most at the interpolation points of g. Also in the former case, for convenience, we say that s *covers* the points $b_j, \ldots, b_{j+k}$ (w.r.t. travel time).

From $(*)$ also follows that any b_j may be covered by at most two sequential segments of f. Thus, there can exist at most $2 \cdot 2 \cdot m + n$ intersections between f and g:

The two incident segments r_1 and r_2 of each point b_j may intersect a segment of f which covers b_j, yielding at most two intersections for b_j. Furthermore r_1 and r_2 may intersect other segments of f, which contain no interpolation point of g, at most once. This may happen at most n times, again due to $(*)$: if r_1 or r_2 intersects a segment s of f, where s contains no point of g, no other segment of g may intersect s.

It follows that the number of intersections between f and g are bounded by $4 \cdot m + n$ and so lie in $\mathcal{O}(n + m)$. □

The running time complexity of Algorithm 3.1 is linear in $n + m$. In the worst case the functions f and g have a number of intersections linear in $n + m$, e.g. one intersection for each segment in f and g (as seen in Figure 3.1c on page 20) resulting in $\frac{n+m}{2}$ intersections. Thus, the running time of the algorithm is optimal in the worst case.

3.2. Link Operation

In time-dependent routing the edge cost functions we use represent the dynamic traffic situation and traversing an edge at different τ may yield different edge cost. Thus, the actual speed at which we traverse the edge varies based on the arrival time at the edge and the definition of the link operation is straightforward.

In the case of electric vehicle routing and tradeoff functions, on the other hand, we wish to compute energy consumption based on travel time. When we traverse two consecutive edges e_1 and e_2 with travel time τ, we may invest τ_1 and τ_2 in e_1 resp. e_2 ($\tau = \tau_1 + \tau_2$). In other words, we spend τ_1 time on e_1 and $\tau - \tau_1$ on e_2. Different values for τ_1 yield different energy consumptions, however we are only interested in minimal consumption. Therefore, we wish to minimize $c(e_1)(\tau_1) + c(e_2)(\tau - \tau_1)$ for $\tau_1 \in [0, \tau]$. And so we obtain the following definition of the link operation:

Definition 3.2. *Link (tradeoff)*

Given tradeoff functions f and g, the link result is defined as

$$(f \circ g)(\tau) := \min_{\tau_1 \in [0,\tau]} f(\tau_1) + g(\tau - \tau_1)$$

For convenience, we denote the linking of edges e_1 and e_2 by $e_1 \circ e_2$ and the linking of an edge e with a path P by $e \circ P$.

After consecutively traversing $e_1 = (u, v)$ and $e_2 = (v, w)$ we obtain the path $P := (u, v, w)$ with cost function $c(e_1) \circ c(e_2)$. We link further edges to P by applying the same definition, as $c(e_1) \circ c(e_2)$ is also a tradeoff function: $c(e_1) \circ c(e_2)(\tau)$ yields the minimal energy consumption when we invest τ in traversing P. In other words, we have the (electric vehicle) tradeoff link equivalent to the time dependant link and may use the definition of the operation in the same manner.

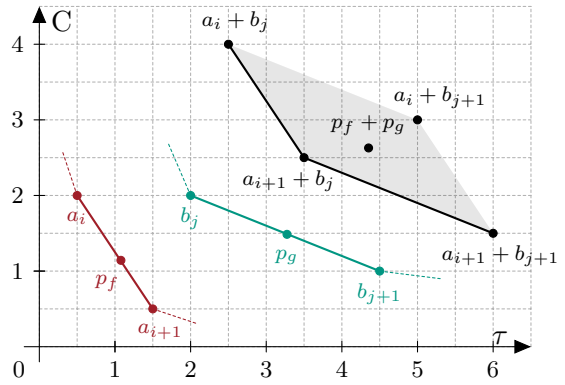


Figure 3.2.: The parallelogram formed by all sums $p_a + p_b, p_a \in \overline{a_i a_{i+1}} \wedge p_b \in \overline{b_j b_{j+1}}$. The polyline from $a_i + b_j$ to $a_{i+1} + b_{j+1}$ via $a_{i+1} + b_j$ dominates points in the parallelogram.

Having defined linking, we now proceed with link computation in the case of tradeoff functions. We first explore the convex sub-case, i.e. linking two convex functions. We then use the convex linking to achieve the linking of an arbitrary function with a convex function. And finally, we compute the link of two arbitrary functions.

3.2.1. Convex Case

As already mentioned, the tradeoff functions at edges are convex. It is therefore meaningful to first examine the link result of two convex edge cost functions. Suppose we have travel time and consumption point $p_1 = (\tau_1, \mathrm{C}_1) \in \mathbb{R}^2$ for e_1, resp. $p_2 = (\tau_2, \mathrm{C}_2) \in \mathbb{R}^2$ for e_2. Furthermore, $c(e_1)(\tau_1) = \mathrm{C}_1$ and $c(e_2)(\tau_2) = \mathrm{C}_2$ or in other words p_1, p_2 lie on $c(e_1)$ resp. $c(e_2)$. Due to the minimum operator in Definition 3.2 on the facing page, the sum $p_1 + p_2$ of any such pair of points must be dominated, as a single point piecewise linear function, by $c(e_1) \circ c(e_2)$.

Let $f := c(e_1)$ and $g := c(e_2)$ be defined by the interpolation points $a, \ldots, 1\mathrm{n}$ resp. $b, \ldots, 1\mathrm{m}$. Further let the minimum and maximum travel times of f and g be $\min_\tau^f, \max_\tau^f$ resp. $\min_\tau^g, \max_\tau^g$. We also set $p_a = p_1$ and $p_b = p_2$. In the case that $\tau_1 \geq \max_\tau^f \wedge \tau_2 \geq \max_\tau^g$ the sum $p_a + p_b$ lies on the horizontal line through $a_n + b_m$. If we choose to traverse e_1 and e_2 at speeds lower then their respective minimum speeds, we do not further lower the energy consumption. Additionally, if $\tau_1 < \min_\tau^f \vee \tau_2 < \min_\tau^g$ then $\mathrm{C}_1 = \infty \vee \mathrm{C}_2 = \infty$ and so $\mathrm{C}_1 + \mathrm{C}_2 = \infty$. In other words, we do not invest enough travel time in f or g and so cannot traverse the linked path. In conclusion, the first point at which we may traverse $e_1 \circ e_2$ is $a_1 + b_1$, and the point after which we no longer benefit from lower speeds is $a_n + b_m$. The linked function $c(e_1) \circ c(e_2)$ therefore has $a_1 + b_1$ as first interpolation point and $a_n + b_m$ as last interpolation point.

We now need to explore the remaining case, i.e. $\tau_1 \in [\min_\tau^f, \max_\tau^f] \wedge \tau_2 \in [\min_\tau^g, \max_\tau^g]$. Let p_a be on the segment $s_a := \overline{a_i a_{i+1}}$ of f, and p_b on the segment $s_b := \overline{b_j b_{j+1}}$ of g, as seen on Figure 3.2. Any sum we may create with two points on s_a resp. s_b lies in the parallelogram formed by the points $a_i b_j := a_i + b_j$, $a_i b_{j+1} := a_i + b_{j+1}$, $a_{i+1} b_j := a_{i+1} + b_j$ and $a_{i+1} b_{j+1} := a_{i+1} + b_{j+1}$, denoted by the grey area of Figure 3.2. That this is a parallelogram is obvious: the segment $\overline{a_i b_j a_i b_{j+1}}$ is parallel to the segment $\overline{a_{i+1} b_j a_{i+1} b_{j+1}}$ as it is the same segment translated by $a_{i+1} - a_i$. The same argument holds for $\overline{a_i b_j a_{i+1} b_j}$ parallel to $\overline{a_i b_{j+1} a_{i+1} b_{j+1}}$ with translation by $b_{j+1} - b_j$. Note that the path formed by the dominating segments of the parallelogram is always convex, due to the convexity of the parallelogram.

Algorithm 3.2: LinkConvex

Input: Convex tradeoff functions f and g defined by interpolation points $a_1, \ldots, a_n$ resp. $b_1, \ldots, b_m$

Output: Linked tradeoff function $\ell := f \circ g$

```
1 ℓ ← ∅
2 i ← 1, j ← 1
  // Sweep functions
3 while i ≤ n and j ≤ m do
      // Add current point to result
4     ℓ.APPEND(a_i + b_j)
      // Pick next point based on slopes
5     if SLOPE(a_i) < SLOPE(b_j) then i ← i + 1
6     else j ← j + 1
```

With this in mind we consider the first point of the linked function, a_1b_1, and the parallelogram formed by $a_1b_1, a_1b_2, a_2b_1, a_2b_2$. We take the first dominating segment r_1 which leads to either a_2b_1 or a_1b_2, depending on whether the slope of $\overline{a_1a_2}$ is less than the slope of $\overline{b_1b_2}$. The first segment of the linking result is then $r_1 = \overline{a_1b_1a_ib_j}$. We then consider a_ib_j in the same fashion and obtain the next segment of the result, r_2. We continue this process iteratively, until we reach the last point of the link result a_nb_m. Algorithm 3.2 outlines the routine and Figure 3.3 on the next page illustrates its application.

Let f and g be convex tradeoff functions, defined by the interpolation points $a_1, \ldots, a_n$ resp. $b_1, \ldots, b_m$. Furthermore, let ℓ be the result of Algorithm 3.2 with input f and g. We now show that ℓ is the link result $f \circ g$.

Lemma 3.3. *The function ℓ is convex.*

Proof. We examine two successive iterations of Algorithm 3.2. In the iteration k we add the point $a_i + b_j$ to ℓ. In the next iteration, $k+1$, we either add $a_i + b_{j+1}$ or $a_{i+1} + b_j$ to ℓ. Thus the segment r_k added to ℓ in iterations k and $k+1$ either has the slope of $\overline{a_ib_ja_ib_{j+1}}$ or $\overline{a_ib_ja_{i+1}b_j}$. In other words, the slope of r_k equals the slope of $\overline{b_jb_{j+1}}$ or $\overline{a_ia_{i+1}}$. In conclusion, ℓ only has slopes which are also found in f and g.

The functions f and g are convex with monotonically increasing slopes. We may thus view the interpolation points of f and g as two lists sorted in ascending order, based on segment slopes. Algorithm 3.2 is equivalent to merging the two lists: in each iteration we "remove" the segment of f or g with lowest slope and add a segment to ℓ with an equal slope. The segments of ℓ are thus sorted by their slope in ascending order, i.e. the slopes of ℓ are monotonically increasing and so ℓ is convex. □

Note that, in further analogy to merging sorted lists, Algorithm 3.2 adds a segment to ℓ for each slope in f and g. I.e. the algorithm does in fact terminate at a_nb_m as stated earlier. The interpolation points a_n and b_m both have slope equal to 0 and so ensure the algorithm adds all slopes present in f and g to ℓ.

Lemma 3.4. *The function $f \circ g$ dominates ℓ.*

Proof. As seen in Algorithm 3.2, the interpolation points of ℓ are of the form $c_k = a_i + b_j$ for some $i \in 1, \ldots, n$, $j \in 1, \ldots, m$ and $k \in 1, \ldots, n+m$. Thus the consumption of ℓ at $c_k = (\tau_k, \mathrm{C}_k)$ is $f(\tau_i) + g(\tau_k - \tau_i)$ where $a_i = (\tau_i, \mathrm{C}_i)$. Per definition of the link operation

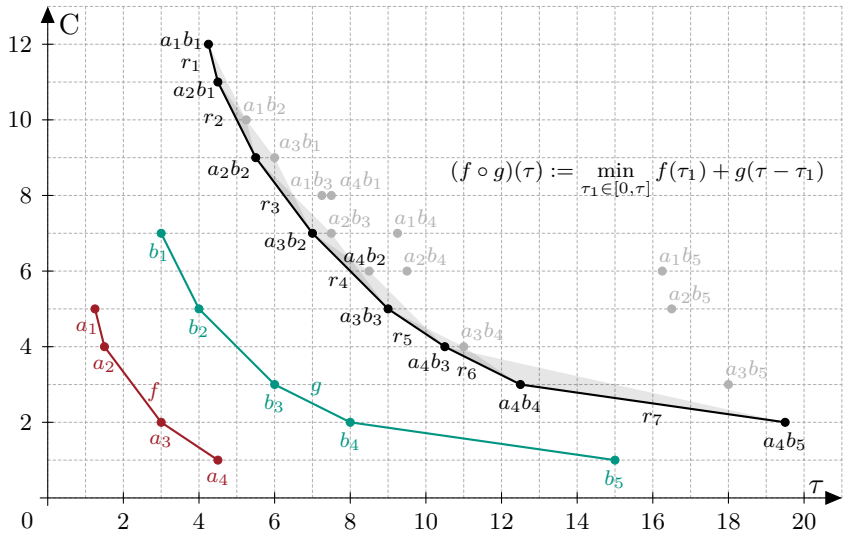


Figure 3.3.: The link result $a \circ b$, defined by the segments r_1 to r_7. The parallelograms of each iteration are denoted by grey area. As seen, we take a segment from the current parallelogram and continue with its other endpoint in the next iteration.

$\mathrm{C}_k \geq (f \circ g)(\tau_k)$ holds, as $(f \circ g)(\tau_k) = \min_{\tau' \in [0,\tau_k]} f(\tau') + g(\tau_k - \tau') \leq f(\tau_i) + g(\tau_k - \tau_i)$, due to $\tau_i \in [0, \tau_k]$. □

Lemma 3.5. *The function ℓ dominates $f \circ g$.*

Proof. To prove that $\ell \propto f \circ g$ we assume that a point $p_a + p_b$ lies below ℓ, for $p_a \in \overline{a_i a_{i+1}}$ and $p_b \in \overline{b_j b_{j+1}}$, and show that such a point cannot exist.

As already seen, $p_a + p_b$ is dominated by a polyline P of two segments from $a_i b_j$ to $a_{i+1} b_{j+1}$, which passes through either $a_i b_{j+1}$ or $a_{i+1} b_j$. As $p_a + p_b = (\tau', \mathrm{C}')$ is not dominated by ℓ the point on P with travel time τ' is not dominated by ℓ. As ℓ is convex and does not dominate P, at least one point on P is not contained in ℓ – otherwise ℓ must pass through P. We choose one such point $a_k b_l \in P$.

We may therefore follow a path P' from $a_k b_l$, which is not dominated by ℓ, by decrementing either k or l in each step, until we reach a point $a_{i'} b_{j'}$ contained in ℓ, e.g. $a_1 b_1$. The first segment s of P' is then either $\overline{a_{i'} b_{j'} a_{i'+1} b_{j'}}$ or $\overline{a_{i'} b_{j'} a_{i'} b_{j'+1}}$. The segment $s = \overline{p_1 p_2}$ ($p_1 = a_{i'} b_{j'}$) has slope lesser than the slope of ℓ at $a_{i'} b_{j'}$, as s is not dominated by ℓ. Due to s being a viable option for the algorithm (at the iteration after adding p_1 to ℓ) the algorithm must pick p_2 owing to the lesser slope of s. Thus the existence of $p_a + p_b$ leads to a contradiction. □

Note that due to the construction of P', each point on P' is a sum of a point on f and a point on g.

Theorem 3.6. *The function ℓ is equal to $f \circ g$.*

Proof. As seen in Lemma 3.4 and Lemma 3.5 $\ell \propto (f \circ g)$ and $(f \circ g) \propto \ell$. This may only be the case if $\ell \equiv f \circ g$. □

The conclusion we draw from Lemma 3.3 and Theorem 3.6 is that $f \circ g$ is convex. Thus, we may link $f \circ g$ with the edge cost function of a further edge, say $h = c(e_3)$. The result

$(f \circ g) \circ h$ will then also be convex, and so on. Furthermore, we know that we can link two convex tradeoff functions in linear time. Algorithm 3.2 computes $f \circ g$ and obviously runs in linear time in the input size. The time complexity is also optimal, as the number of interpolation points of $f \circ g$ is linear in the size of f and g (e.g. Figure 3.3 on the preceding page).

3.2.2. Arbitrary Case

The arbitrary case of linking tradeoff functions is far less elegant then the convex case. We require the linking of a non-convex function with a convex one due to the merge operation. The merge result of two convex functions is not necessarily convex, e.g. Figure 3.1c on page 20. As we use Algorithm 2.2 on page 12 to compute profiles, our tradeoff functions will always be linked with convex edge cost tradeoff functions. Thus we first explore the case where one function is convex, and the other is not.

Half-convex Linking. As usual, let f and g be tradeoff functions with interpolation points $a_1, \ldots, a_n$ resp. $b_1, \ldots, b_m$. Furthermore let f be convex. While g is not necessarily convex, it is composed of convex sub-functions $g_1, \ldots, g_k$. In the most extreme case, g is concave and so is composed of $m-1$ sub-functions, namely the segments of g (i.e. $\overline{b_j b_{j+1}}$). Thus we first identify $g_1, \ldots, g_k$ by sweeping $b_1, \ldots, b_m$. Whenever the slope of b_j is greater than the slope of b_{j+1} the current convex sub-function ends, and the next one begins. Note that we examine the convex sub-functions g_j as "standalone" piecewise linear functions, i.e. the last point of g_j has slope equal to 0. This is despite the fact that g_{j+1} begins at the same point, with a non zero-slope.

We may then use Algorithm 3.2 on page 24 to compute $f \circ g_1, \ldots, f \circ g_k$. For each τ we take $\min_{i \in j, \ldots, k} (f \circ g_j)(\tau)$ for the value of $\ell(\tau)$, resulting in $\ell \equiv f \circ g$.

Figure 3.4 on the facing page shows an example of the half-convex linking. To compute the minimum function of $f \circ g_1, \ldots, f \circ g_k$ we may consecutively apply the merge operator, which we discuss in Section 3.1 on page 20. However, we may exploit a property of the link operation to avoid the full merging. Furthermore, we avoid some of the computations during the linking of f with the convex sub-functions g_j.

Namely, for $j \in 1, \ldots, k-1$, $f \circ g_j$ and $f \circ g_{j+1}$ have exactly one intersection (Lemma 3.7). We also sweep $f \circ g_j$ and $f \circ g_{j+1}$ from minimum to maximum travel time, as in the merge operation. Therefore, we stop the sweeping as soon as we find the sole intersection point p. The result of the merge is then $f \circ g_j$ until p, concatenated with $f \circ g_{j+1}$ after p. Obviously, computing all of $f \circ g_j$ is wasteful – we need only the segments before and directly after p. Thus, we first compute $f \circ g_k$. We then link f with g_{k-1}, but stop the linking as soon as $f \circ g_{k-1}$ intersects $f \circ g_k$. We then proceed with $f \circ g_{k-2}$ and so on. See Algorithm 3.3 on page 28 for the outline of the routine.

Although this method does not reduce the asymptotic computational complexity, we avoid unnecessary linking and reduce the sweep steps during the merges. We now prove the sole intersection property and then prove the correctness of the half-convex linking method. Similar to the convex linking, let ℓ be the result of Algorithm 3.3 with input tradeoff functions f and g (where f is convex).

Lemma 3.7. *Let f and g be tradeoff functions, where f is convex and g is composed of convex sub-functions $g_1, \ldots, g_k$. Then $f \circ g_j$ and $f \circ g_{j+1}$ have exactly one intersection, for $j \in 1, \ldots, k-1$.*

Proof. Let $j \in 1, \ldots, k-1$, $e \equiv f \circ g_j$ and $h \equiv f \circ g_{j+1}$. Furthermore let the interpolation points of f be $a_1, \ldots, a_n$.

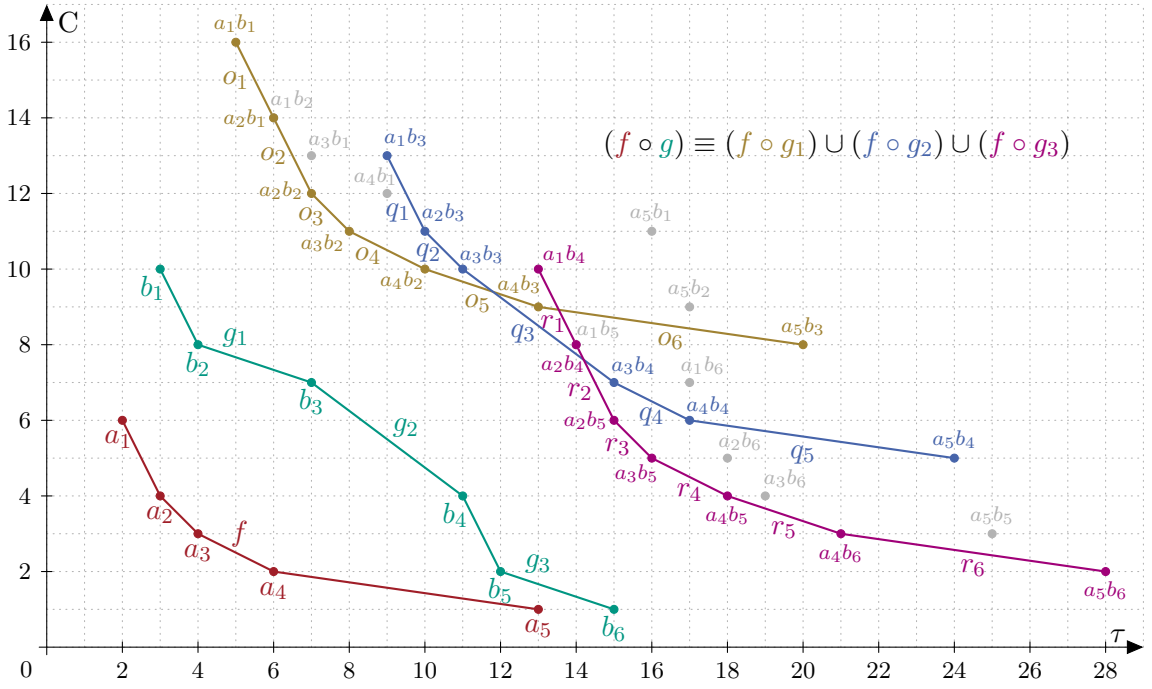


Figure 3.4.: Linking f with the convex sub-functions of g, namely g_1, g_2 and g_3. The linked functions $f \circ g_1$ is composed of the segments $o_1, \ldots, o_6$. Likewise, $f \circ g_2$ and $f \circ g_3$ are defined by the segments q_i resp. r_j. The minimum of the three linked functions is the link result $f \circ g$. We observe that $f \circ g_1$ and $f \circ g_2$ have exactly one intersection point. Furthermore, this is also true for $f \circ g_2$ and $f \circ g_3$.

We must first prove that e and h have an intersection. As g_j and g_{j+1} are consecutive sub-functions of g, they share a point p on g: the last point of g_j and the first of g_{j+1}. Thus, both e and h must dominate $p + a_1$ due to the link definition. As $p + a_1$ is the first point of h, e must have energy consumption values below that of $p + a_1$. Furthermore the minimum energy consumption of h lies below that of e and the minimum travel time of e is less then that of h (g is monotonic). Therefore, the energy consumption of the last point of e must lie in the energy consumption interval of h. Likewise the travel time of the first point of h must lie in the travel time interval of e. Thus e intersects h. Note that this intersection may lie on the horizontal line of e after its last interpolation point. This occurs whenever the two last points of g_j lie on a horizontal segment, i.e. g_j ends with a horizontal segment.

We now prove that there is at most one intersection between e and h. We take advantage of f's convexity and the fact that both e and h result from linking a convex tradeoff function with f. Consider a vertical line l at the minimum travel time $\min_\tau$ of h, see Figure 3.5 on page 29. Let τ be the travel time of the first intersection between e and h and let $x \in \mathbb{R} : x \geq \tau$. We show that the slope of e at x is greater or equal than the slope of h again at x. If this holds, then obviously e and h have no further intersections: a second intersection requires that e has a slope lower than that of h in some sub-interval of (τ, ∞).

Consider $p_a \in e$ and $p_b \in h$ both with travel time τ. At p_b we have at most used a portion of f with travel time x, whereas at p_a we have used atleast a x travel time portion of f. The reason for the latter statement is simple: the travel time interval between the first points of e and h is equal to exactly the travel time interval of g_j. The function $f \circ g_j$ begins with a_1 plus the first point of g_j, and $f \circ g_{j+1}$ begins with a_1 plus the last point of g_j.

Algorithm 3.3: LinkHalfConvex

Input: Tradeoff functions f and g defined by interpolation points $a_1, \ldots, a_n$ resp. $b_1, \ldots, b_m$, with f convex and g composed of convex sub-functions $g_1, \ldots, g_k$

Output: Linked tradeoff function $\ell := f \circ g$

```
1  ℓ ← ∅
2  forall j ∈ 1,...,k do
       // Compute h_j := f ∘ g_j until first intersection with ℓ
3      s_j, e_j ← indices of first resp. last point of g_j in g, h_j ← ∅
       // Link sweep, each iteration followed by partial sweep of ℓ
4      i' ← 1, j' ← s_j
5      while i' ≤ n and j' ≤ e_j do
6          p ← a_{i'} + b_{j'}
7          h_j.APPEND(p)
8          if SLOPE(a_{i'}) < SLOPE(b_{j'}) then i' ← i' + 1
9          else j' ← j' + 1
           // Cut ℓ until summed travel time τ of p
10         while ℓ.max_τ ≤ p.τ do
               // Intersection test with segment of h_j ending at p and
                  first remaining segment of ℓ
11             c ← last segment of h_j, d ← first segment of ℓ
12             ℓ.REMOVE FIRST()
13             if c intersects d then
14                 h_j.REMOVE LAST()
15                 h_j.APPEND(INTERSECTION(c,d))
                   // Link sweep done after first found intersection
16                 go to 18
       // No inner intersection ⇒ horizontal line of h_j intersects ℓ
17     ℓ.CUT(≥ h_j.min_C)
       // Insert h_j at the front of ℓ
18     ℓ.PREPEND(h_j)
```

Due to our observations in Lemma 3.3, we can draw the following conclusion. The slope at p_a is at most m and the slope p_b is atleast m, where m is the slope of f at travel time x. As this holds for any x greater than τ, the slope of e is greater or equal than the slope of h in $[\tau, \infty)$. Therefore, $f \circ g_j$ and $f \circ g_{j+1}$ may not intersect in (τ, ∞) and so have only one intersection at τ. □

Theorem 3.8. *Let f and g be tradeoff functions, where f is convex and g is composed of convex sub-functions $g_1, \ldots, g_k$. Then the function ℓ equals $f \circ g$.*

Proof. First note that due to the link definition, the functions $h_1 := f \circ g_1, \ldots, h_k := f \circ g_k$ dominate all points $p_a + p_b$ where $p_a \in f$ and $p_b \in g$. It is therefore enough to show that ℓ dominates $h_1, \ldots, h_k$. We know that $(f \circ g) \propto \ell$ due to the definition of the link operation and Lemma 3.4.

We start with the link h_k, which obviously dominates h_k. The next step of Algorithm 3.3 merges h_{k-1} with h_k (Lemma 3.7) and the result so far dominates both functions. In the

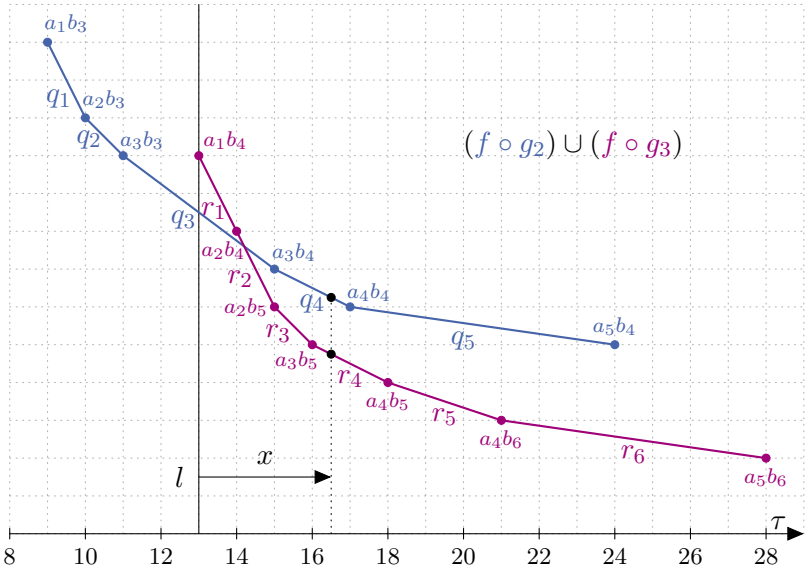


Figure 3.5.: Each pair $h_j := f \circ g_j$ and $h_{j+1} := f \circ g_{j+1}$ (for $j \in 1, \ldots, k-1$) intersect exactly once. To prove this, we show that the slope of h_j is equal or greater than that of h_{j+1} in the interval $[\min_\tau^{h_{j+1}}, \infty)$.

next iteration the algorithm merges the "surviving" portion of h_{k-2} with h_{k-1}. Again due to Lemma 3.7 an intersection is ensured. The portion of h_{k-2} with travel time greater than the one of this intersection is, of course, dominated by h_{k-1}. It is therefore also dominated by $h_{k-1} \cup h_k$ which was computed in the previous iteration. We may further apply the same observation, until we reach h_1. □

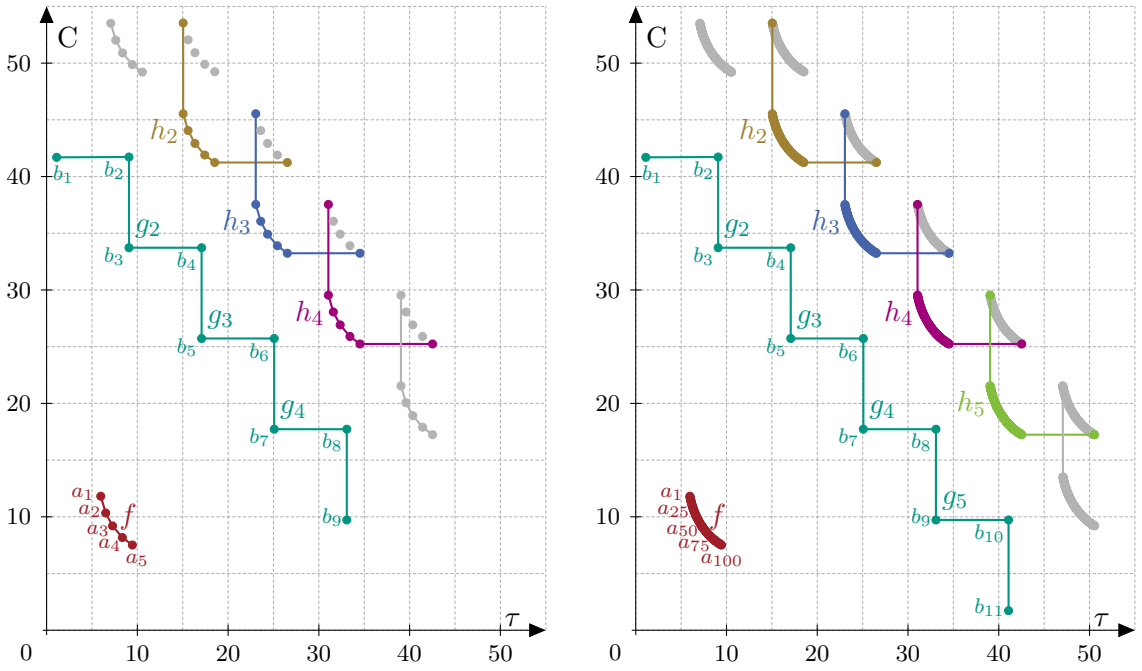
The time complexity of the half-convex link operation is in $\mathcal{O}(\sum_j (n + m_j)) = \mathcal{O}(n \cdot k + m)$, where m_j is the number of interpolation points of g_j ($j \in 1, \ldots, k$). In the worst case, g is concave and so $k = m - 1$ and $m_j = 1$. The worst case complexity thus lies in $\mathcal{O}(n \cdot m)$, which is significantly worse than the linear time required in convex linking. We now prove that the worst case running time is also in $\Omega(n \cdot m)$ and therefore in $\Theta(n \cdot m)$. To show this, we first construct a case where the linked function has a number of points in $\mathcal{O}(n \cdot m)$. By extension, any algorithm which computes this link must invest $\Omega(n \cdot m)$ operations.

Theorem 3.9. *Let f and g be tradeoff functions with n resp. m interpolation points, where f is convex and g is composed of convex sub-functions $g_1, \ldots, g_k$. Then the worst case of computing $f \circ g$ lies in $\Omega(n \cdot m)$.*

Proof. We first construct g as a staircase function and f as an approximation of a curve, both monotonic. We start and end g with a horizontal resp. vertical segment. In-between, every two points we add to g result into a new convex sub-function and so $2 \cdot k = m - 1$. See Figure 3.6a for an example of f, g and the linked functions $f \circ g_1, \ldots, f \circ g_k$.

While g may be any staircase tradeoff function, the travel time and consumption intervals of f must be at most as wide as those of g_j, $j \in 2, \ldots, k-1$. In this manner a linear portion of each link $f \circ g_j$ is contained in the result $f \circ g$.

On Figure 3.6b we see that we may obviously extend g with further "steps" and always produce further points in the result. In addition, we may also increase the number of interpolation points in f and still obtain a result with $\mathcal{O}(n \cdot m)$ points. We only refine the curve approximation represented by f, without increasing the span of f.



(a) (b)

Figure 3.6.: A constructed case where each $f \circ g_j$ introduces $\mathcal{O}(n)$ points to the link result, for $j \in 1, \ldots, k, k \in \mathcal{O}(m)$. Here n and m are the sizes of f resp. g. The number of convex sub-functions of g is k. As seen on the right, the staircase function g may be extended by additional steps and f by additional points. The resulting size complexity of the link function thus lies in $\mathcal{O}(n \cdot m)$.

As $2 \cdot k = m - 1$, k is in $\mathcal{O}(m)$. The resulting function $f \circ g$ has $\mathcal{O}(n \cdot k)$ points and therefore also $\mathcal{O}(n \cdot m)$ points. Thus, any algorithm which computes $f \circ g$ must run in $\Omega(n \cdot m)$ in the worst case. □

Non-convex Linking. While Algorithm 3.3 on page 28 is sufficient for the standard profile algorithm, we also require linking two arbitrary tradeoff functions. Though edge cost functions are convex, the profile from s to t is generally not convex due to the merge operation. If we wish to apply the CH algorithm (or any other shortcut based algorithm), the cost functions of the shortcuts are exactly such profiles. We must therefore link a profile at some vertex u with a shortcut from u. Thus, we now explore the arbitrary linking case.

We may split the cost functions f and g into convex sub-functions $f_1, \ldots, f_l$ resp. $g_1, \ldots, g_k$, in the same way we split g before applying Algorithm 3.3. To dominate each $p_a + p_b$, with p_a and p_b on f resp. g, it is enough to compute all pairs $f_i \circ g_j$ and merge the resulting functions. As we take the overall minimum of all link operations, the resulting function is ensured to dominate $f \circ g$ and so must be equal to $f \circ g$. See Figure 3.7a on the next page for an example.

A less sophisticated routine computes $f_1 \circ g, \ldots, f_k \circ g$ and merges the resulting functions, see Figure 3.7b. As it is enough to merge the pairs $f_i \circ g$ and $f_{i+1} \circ g$, we do not require a complicated merging procedure such as a sweep-line [SH76] based algorithm. Furthermore we may spare some of the linking, due to our observations in the half-convex linking case: computing $f_i \circ g$ will stop linking $f_i \circ g_j$ as soon as an intersection with $f_i \circ g_{j+1}$ is found.

Algorithm 3.4 on page 32 outlines the arbitrary linking algorithm. Let $h_i := f_i \circ g, i \in 1, \ldots, l$. Similar to the half-convex linking, we compute the functions h_i in reverse order, starting

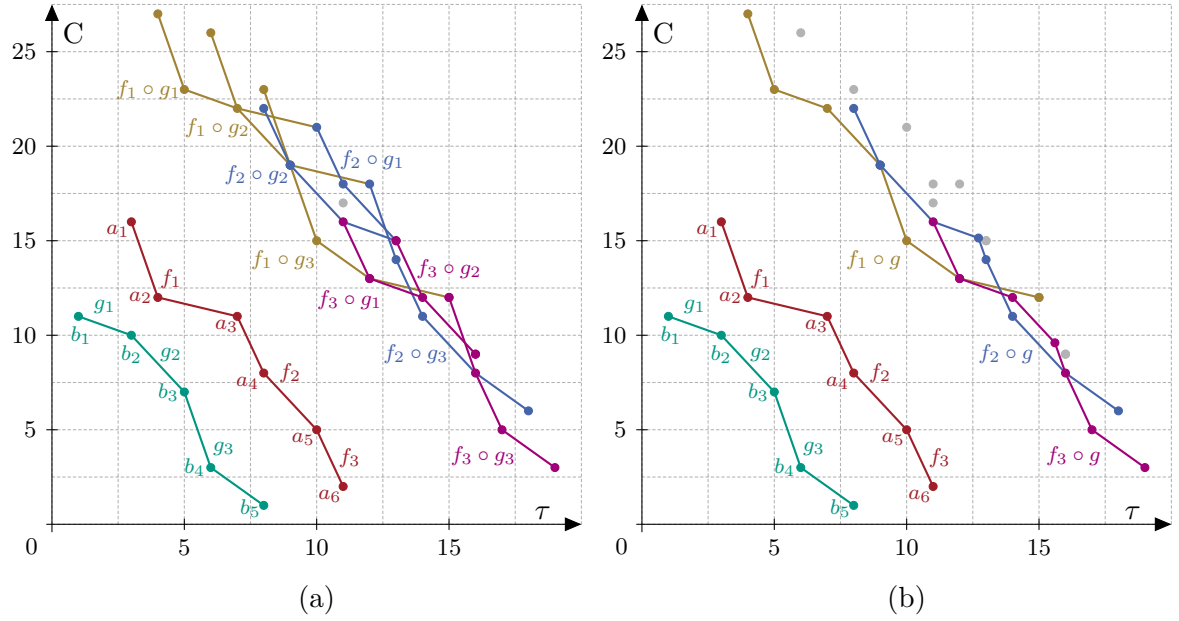


Figure 3.7.: Two alternatives of computing $f \circ g$ for arbitrary tradeoff functions f and g. First, we may link all pairs $f_i \circ g_j$ and merge the result, as seen on the left. Or second, we may use the half-convex linking to compute $f_i \circ g$ and then merge $f_i \circ g$ with the already linked $f_{i+1} \circ g$.

with h_l. In the iteration i in which we compute h_i we have already computed the minimum h'_i of $h_{i+1}, \ldots, h_l$, i.e. $h'_l = \emptyset$ and $h'_o = h_o \cup h'_{o+1}$ for $o \in 1, \ldots, l-1$. As seen in Figure 3.7b, consecutive h_i and h_{i+1} may have multiple intersections ($f_2 \circ g$ and $f_3 \circ g$). Thus, we must compute all of h_i before merging with h'_i, unlike the half-convex linking approach.

We now consider an individual merge $h_i \cup h'_i, i \neq l$ (line **4**). As $\min_{\mathrm{C}}^{f_i} < \min_{\mathrm{C}}^{f_{i+1}}$, the minimum energy consumption of h_i is strictly less than that of h_{i+1}. Therefore the portion of h'_i below $\min_{\mathrm{C}}^{h_{i+1}}$ will not be affected by the merge and may be ignored. Thus, we stop the merge algorithm as soon as the current point of h'_i requires strictly less consumption than the last point of h_i. We may test this condition between lines **6** and **7** of Algorithm 3.1 on page 21.

Note that as in the half-convex case, the last intersection may lie on the horizontal line of h_i. Similarly, even if the current point of h'_i requires strictly more travel time than the last point of h_i we can not stop the merging routine.

Theorem 3.10. *Given arbitrary tradeoff functions f and g, the function ℓ computed by Algorithm 3.4 equals $f \circ g$.*

Proof. The link $f \circ g$ dominates ℓ per definition, as all points on ℓ result from the linking of some $f_i \circ g_j$. Furthermore, the minimum of all $f_i \circ g_j$ dominates $f \circ g$ due to Theorem 3.6.

For a specific i, each separate link $f_i \circ g$ in ℓ dominates $f_i \circ g_j, j \in 1, \ldots, k$ as proven in Theorem 3.8. Thus ℓ dominates all $f_i \circ g_j$ and therefore $f \circ g$. □

We are also interested in the running time of Algorithm 3.4. Due to our observations in the half-convex case, the running time of each link h_i lies in $\mathcal{O}(n_i \cdot m)$. Here each n_i is the number of interpolation points of f_i. The total cost for all links therefore lies in $\mathcal{O}(\sum_i n_i \cdot m) = \mathcal{O}(n \cdot m)$ (as $\sum_i n_i = n$). I.e. the linking cost is not asymptotically worse

Algorithm 3.4: LinkArbitrary

Input: Tradeoff functions f and g, f composed of convex sub-functions $f_l, \ldots, f_l$
Output: Linked tradeoff function $\ell := f \circ g$

```
1 ℓ ← ∅
2 forall i ∈ 1, ..., l do
      // Compute h_i := f_i ∘ g
3     h_i ← LinkHalfConvex(f_i, g)
      // Compute h'_{i-1} := h_i ∪ ℓ until min_C^{h_i}
4     h'_{i-1} ← Merge*(h_i, ℓ)
5     ℓ.cut(≥ h'_{i-1}.min_C)
6     ℓ.prepend(h'_{i-1})
```

than in the half-convex case. However, we must also take the merging at each iteration into account.

Each h_i has $\mathcal{O}(n_i + m)$ interpolation points due to Lemma 3.1. Furthermore $h_i \cup h'_i$ has $\mathcal{O}(\sum_{o=i}^{l}(n_i + m))$ points. Thus, if we merge $h_i \circ h'_i$ wholly, the total running time complexity lies in $\mathcal{O}(\sum_{i=1}^{l}\sum_{o=i}^{l}(n_i + m)) = \mathcal{O}(\sum_{i=1}^{l} i \cdot (n_i + m))$. In the worst case f is concave, so $l = n - 1$ and $n_i \in \mathcal{O}(1)$. The running time then lies in $\mathcal{O}(m \cdot \sum_{i=1}^{n} n_i) = \mathcal{O}(m \cdot n^2)$, which is obviously not feasible – a sweep-line based routine would be asymptotically superior. It is therefore important to show that the running time of Algorithm 3.4 is at most in $\mathcal{O}(\log m \cdot n \cdot m)$, and not in $\mathcal{O}(m \cdot n^2)$.

Proving that the costs of each merge lie in $\mathcal{O}(n_i + n_{i+1} + m)$, i.e. linear in the number of interpolation points of h_i and h_{i+1}, is sufficient. The linking cost will then "cover" the merge cost. Here two observations are important. First, the merge in iteration i stops as soon as the last point p of h_i is reached. In other words, in order to exceed $\mathcal{O}(n_i + m)$ in iteration i a non-constant number of points in h'_i must have greater energy consumption than p. And second, whenever we merge $h_i \cup h'_i$ only the dominating portions remain. For the total merge complexity to exceed $\mathcal{O}(n \cdot m)$, the dominated portion r of h_i must be replaced by a super-linear (in the size of r) number of points in ${h_i}'$. Furthermore, the replacing points must not be "paid for" by portions of previous $h_j, j \geq i$. I.e. for $o \in \mathcal{O}(l)$ an x_i number of points of each $h_i, i \in o, \ldots, l$ must satisfy the following conditions for each $j \in o, \ldots, l$:

- are not dominated by h_j
- have energy consumption greater than $\min_C^{h_j}$
- no mutual constant bounds x_j

Using the above information plus a slope argument (as in Lemma 3.7) should suffice to prove that the running time of Algorithm 3.4 is in $\mathcal{O}(n \cdot m)$. However, we were unable to produce a formal proof. We were also unable to construct a case where the complexity exceeded $\mathcal{O}(n \cdot m)$ and so believe that such a proof does exist.

It follows from Theorem 3.9 that the worst case of the arbitrary linking operation is also in $\Omega(n \cdot m)$. Here, we use the fact that any algorithm which links two non-convex functions may be used to link a convex and a non-convex function. Thus, if the running time of Algorithm 3.4 is indeed in $\mathcal{O}(n \cdot m)$, then the algorithm is optimal in the worst case.

Algorithm 3.5: DOMINATES

Input: Tradeoff functions f and g defined by interpolation points $a_1, \ldots, a_n$ resp. $b_1, \ldots, b_m$

Output: **true** iff $f \propto g$

```
1  if f.min_τ > g.min_τ or f.min_C > g.min_C then
2      return false
3  a_{n+1} ← (∞, f.min_C), b_{m+1} ← (∞, g.min_C)
4  i ← 1, j ← 1
5  while i ≤ n and j ≤ m do
       // Interpolate a_i a_{i+1}, test if b_j above
6      if b_j.τ ∈ [a_i.τ, a_{i+1}.τ] and b_j below a_i a_{i+1} then
7          return false
       // Interpolate b_j b_{j+1}, test if a_i below
8      if a_i.τ ∈ [b_j.τ, b_{j+1}.τ] and a_i above b_j b_{j+1} then
9          return false
10     if a_{i+1}.τ < b_{j+1}.τ then  i ← i + 1
11     else  j ← j + 1
12 return true
```

3.3. Domination Test

Finally, we require a procedure to test whether a tradeoff function f dominates another, g. As usual, let f and g be defined by interpolation points $a_1, \ldots, a_n$ resp. $b_1, \ldots, b_m$.

We test if $f \propto g$ with Algorithm 3.5. We first check whether g allows less travel time or energy consumption than f and if so return $f \not\propto g$. We then sweep the interpolation points of f and the segments of g, testing if a_i lies below $\overline{b_j b_{j+1}}$ if the travel time of a_i lies in the travel time interval of $\overline{b_j b_{j+1}}$. Here, $a_i = (\tau, \mathrm{C})$ is below / above $r := \overline{b_j b_{j+1}}$ if C is less / greater than or equal to r interpolated at τ. We do the same for the interpolation points of g, however we test whether each b_j lies above the respective segments of f. We return $f \not\propto g$ on the first failed test, otherwise $f \propto g$ holds.

Theorem 3.11. *Given tradeoff functions f and g, Algorithm 3.5 returns* **true** *iff $f \propto g$.*

Proof. If Algorithm 3.5 returned **false**, then either g allows less travel time or energy consumption than f, there is an a_i above g or a b_j below f. This implies $f \not\propto g$.

For the other direction of the equivalence, assume all a_i lie below g, all b_j above f, $\min_\tau^f \leq \min_\tau^g$ and $\min_\mathrm{C}^f \leq \min_\mathrm{C}^g$. We now show that no intersection between f and g exists in $(\min_\tau^f, \infty)$. As the interpolation points of g lie above those of f and f dominates g in the travel time interval $(-\infty, \min_\tau]$, $f \propto g$ follows.

Assume an intersection p between f and g exists, where the travel time of p lies in $(\min_\tau^f, \infty)$. Thus p must lie on segments $c := \overline{a_i a_{i+1}}$ and $d := \overline{b_j b_{j+1}}$ with $i \in 1, \ldots, n-1$ and $j \in 1, \ldots, m-1$, as a_n and b_m have horizontal slopes: f and g can be at most overlapping from a_n resp. b_m on. If b_{j+1} lies in the travel time interval of c, then b_{j+1} must lie below c due to g being monotonic. Thus b_{j+1} must have travel time greater than that of a_{i+1}.

However, this would mean that a_{i+1} lies above $r := \overline{p b_{j+1}}$ as the travel time of a_{i+1} lies in the travel time interval of r. The segment r is a sub-segment of d and so a_{i+1} lies above d. This contradicts our assumption that no point of f lies above g. □

The running time complexity of Algorithm 3.5 is linear in $n + m$. This is also optimal in the worst case, as we may push a random point of g below f (or a point of f above g) before running any deterministic online algorithm. The algorithm is therefore obliged to check all of a_i and b_j.

4. Basic Approach

In Chapter 3 we devised routines for merging, linking and domination tests. We may now proceed with profile algorithms. Throughout this chapter, let $G = (V, E)$ be a directed weighted graph where the edge weights are convex trade-off functions. Furthermore, we assume the travel time cost of edges is positive and that G has no loops.

First, we directly apply the profile operations as seen in Algorithm 2.2 on page 12, which computes time-dependent profiles. In this manner we obtain a label correcting algorithm, where our only degree of freedom is the choice of priority queue keys.

We then alter this algorithm to gain the label setting property. We achieve this by combining the standard time-dependent profile search and multi-criteria Dijkstra. Furthermore we offer path extraction and speed directions for the altered algorithm.

4.1. Label Correcting

As mentioned, we adopt the profile search used in time-dependent routing. The label $c(v)$ at each vertex $v \in V$ is a trade-off function , which is empty at the start of the algorithm. The source vertex s is initialized with a single-point function, say 0 travel time and energy consumption, and is added to the priority queue. Relaxing an edge $e = (u, v)$ amounts to half-convex linking (Algorithm 3.3 on page 28), as the tradeoff function at e is convex and the function at u is arbitrary. The resulting function $h_e := f_u \circ c(e)$ and f_v are then tested for dominance with Algorithm 3.5 on page 33. If one dominates the other, f_v is set to the dominating function. Otherwise, h_e is merged into f_v by applying Algorithm 3.1 on page 21. In either case, if f_v is changed, v is added to the priority queue or if already present its key is updated. In each iteration we pull the vertex with minimal key from the queue and relax its outgoing edges. The algorithm terminates when the priority queue is empty, i.e. when the functions $f_v, v \in V$ represent optimal tradeoff functions.

As in time-dependent routing, multiple paths to a single vertex v may yield non-dominating functions. Each such function is propagated at different times and so v is visited multiple times, i.e. the algorithm is label correcting. Each time the tradeoff function at v is "corrected", the improvement must be propagated. The main drawback of the algorithm is that the propagation is done with the whole function. Relatively small (say single point) improvements therefore have the same impact on the running time as whole-scale ones.

The impact of this drawback on the running time is significant (Chapter 6). While the label setting approach amends this shortcoming, the standard profile algorithm remains

viable. This is the case as we may set arbitrary keys for the priority queue – an option which is no longer available in the label setting algorithm. Combined with target pruning, which we explain in the next paragraph, choice keys lead to better (compared to label setting) running times (Chapter 6).

Note that we only consider the profile algorithm with target pruning, as we are interested in s–t queries. Without target pruning we compute the profile tradeoff function at each vertex of G, which is unnecessary in our scenario. As explained in the following paragraph, target pruning requires the non-negative reduced edge costs offered by A*. We explain the application of A* in Chapter 5, for now only the gained non-negative reduced costs are important. Note that we do not change edge cost functions with the application of A*: we only shift the priority queue keys.

Standard Enhancements. Target pruning is a standard technique used in s–t shortest path queries, e.g. the multi-criteria Dijkstra algorithm. Whenever we pull a vertex v from the priority queue, we may prune v if $f_t \propto f_v$ holds. Here, pruning means we ignore v and do not relax its outgoing edges. The reasoning behind target pruning is simple: as long as the edge costs of G are non-negative and $f_t \propto f_v$, reaching the target vertex t via v will result into a function that is also dominated by f_t.

By using target pruning we may avoid visiting the whole graph G during the search, similar to the stopping criterion of Dijkstra's algorithm. However, the technique requires non-negative edge costs. While the travel time costs of the tradeoff functions are always positive, recuperation results in some edges with negative energy consumption. It is therefore possible, that by not pruning v we may reach t with lower consumptions than the ones in f_v. In other words, f_v may be dominated by f_t, but the path to t via v may still be viable due to downhill edges on the path. So in order to use target pruning, we also require the non-negative reduced edge costs offered by the A* algorithm.

A further common improvement is *time-stamping*. When running multiple queries, Algorithm 2.2 will initialize each f_v anew before the main loop. Thus, even if the search does not visit all vertices of G due to target pruning, we still have linear costs in the number of vertices. This is remedied by maintaining timestamps. Whenever an edge to v is relaxed and v has an outdated stamp, f_v is initialized and the stamp at v is updated. The initialization phase of Algorithm 2.2 may then be skipped.

Priority Queue Keys. The label correcting property of the algorithm enables the use of any priority queue key for a function f_v, as long as we apply A*. Even if we use target pruning, we do not require monotonically increasing keys (w.r.t. algorithm iterations). The reason for this is simple: the algorithm may only terminate if there are no improvements to propagate, or such improvements are dominated by f_t. Furthermore, due to A* the reduced edge costs are non-negative i.e. the improvements can only degrade during the propagation. Thus the order in which we process the vertices is irrelevant for the correctness of the algorithm, and this order is determined by the priority queue keys.

The order of processing, however, may greatly reduce (or increase) the running time of the algorithm (Chapter 6). Due to target pruning combined with A* vertex potentials (Chapter 5), the sooner we obtain a "good" function at t the more vertices we might be able to prune. This in turn reduces the number of queue pulls performed by Algorithm 2.2.

According to Baum et al. [BDHS+14], computing the Pareto set of tradeoff points performs best when using the point p of $\mathcal{B}(v)$ with maximum travel time $\max_\tau$ and minimum energy consumption $\min_C$ as the key for v. As p is two dimensional, a linear combination of $\max_\tau$ and $\min_C$ is used. We may therefore use the last point of the tradeoff function f_v in a similar fashion.

When computing lower bound functions in a related context, Zündorf [Zün14], observes that such keys may not reflect changes during a merge. Take the relaxation of $e = (u, v)$ as an example. If h_e dominates f_v in the interval $[\min_\tau^h, \max_\tau^{f_v})$ but not at $\max_\tau^{f_v}$, then the priority queue key of v will not change even though the function at v did. Instead, they use the first point where f_v is changed during the merge. Further, after relaxing the outgoing edges of u, they set the priority queue of u to ∞ – any change due to f_u will be propagated by the neighbours of u and so u is no longer important (until a new change in f_u occurs).

We adopt the approach of Zündorf [Zün14], however we use the last point where f_v changed and not the first. This adaptation is, of course, based on the findings of Baum et al.

Overcharging and Undercharging. Whenever we relax an edge $e = (u, v)$ and compute h_e undercharging or overcharging may occur. As the resulting f_v is a tradeoff function, it maps travel time to energy consumption. We define energy consumption (Chapter 2) as $\mathrm{M} - \mathrm{SoC}$, i.e. the amount required to charge the battery from the current state of charge SoC to the maximum capacity M.

Overcharging therefore occurs when the consumption drops below 0 as we may not charge the battery higher than M. Similarly, undercharging occurs if the consumption is higher then M: SoC is non-negative at all times.

To handle the two events we must thus cut any values of h_e that lie (strictly) below 0 or above M. As the initial battery capacity is known at the algorithm start, ensuring no tradeoff function values lie outside $[0, \mathrm{M}]$ after each link is sufficient to handle overcharging and undercharging.

4.2. Label Setting

We now wish to introduce the label setting property to the profile algorithm. Although our label setting approach does not yield lower query times, it may still be used in other contexts. For instance, it may be combined with the hyperbolic functions in [Zün14]. To obtain the label setting property, we must ensure that portions of f_v at v are processed only once. We must therefore split f_v in some way and process the split portions one at a time. Then we ensure that these portions are not processed multiple times.

Instead of maintaining a single function f_v at v, we maintain a bag of functions $\mathcal{B}(v)$. The whole $\mathcal{B}(v)$ then represents f_v, and we may process the elements $\mathcal{B}(v)$ one at a time. We choose convex sub-functions as a splitting criterion, as we then may link ℓ in linear time with $c(e)$ for any edge $e \in E$.

For the outline of the approach we adapt the multi-criteria Dijkstra, see Algorithm 2.3 on page 15. As the multi-criteria algorithm uses k-dimensional points and not functions, we must replace some of its operations. Given a label $\ell \in \mathcal{B}(v)$ and an edge $e = (v, w)$, we first substitute the vector sum (line **11**) with the convex linking operation (Algorithm 3.2 on page 24) which computes $\ell \circ c(e)$. For the domination test and the merge operation we adjust Algorithm 3.5 resp. Algorithm 3.1. In other words, we use the appropriate operations from Chapter 3.

An s–t query therefore functions as follows. We set $\mathcal{B}(s) = \{\ell_0\}$, where ℓ_0 is a single-point zero tradeoff function (i.e. $(0, 0)$) and insert s in the priority queue with key 0. In each iteration we pull an unprocessed label ℓ from $\mathcal{B}(u)$, where u is the vertex at the head of the priority queue. We mark ℓ as processed, and if $\mathcal{B}(u)$ contains only processed labels, then we remove u from the queue. If $\mathcal{B}(u)$ does contain unprocessed labels, we update the priority queue key of $\mathcal{B}(u)$. For each edge $e = (u, v)$ we then compute $h_e^\ell := \ell \circ c(e)$ and test whether $\mathcal{B}(v) \propto h_e^\ell$. If not, we merge h_e^ℓ into $\mathcal{B}(v)$ and update the priority queue w.r.t.

v. After the merge the segments $\mathcal{B}(v)$ dominated by h_e^ℓ are replaced with the dominating segments of the latter. Furthermore, $\mathcal{B}(v)$ is again cut into convex sub-functions, each represented by a label. And last, the algorithm terminates as soon as the queue is empty.

We now proceed with the adjustments required to gain the label setting property. Our goal is to ensure that whenever we mark a label $\ell \in \mathcal{B}(v)$ as processed, all labels in $\mathcal{B}(v)$ with less minimum travel time than $\min_\tau^\ell$ are already processed (in previous iterations).

Bag Order and Priority Queue Keys. We first impose an ordering on each $\mathcal{B}(v), v \in V$ based on the minimum travel time of the labels. In other words, if $\mathcal{B}(v) = \{\ell_1, \ldots, \ell_n\}$ then $\min_\tau^{\ell_i} < \min_\tau^{\ell_{i+1}} \wedge \max_\tau^{\ell_i} \leq \min_\tau^{\ell_{i+1}}, i \in 1, \ldots, n-1$ must hold at all times. Once again, the labels of $\mathcal{B}(v)$ are the convex sub-functions of the function f represented by $\mathcal{B}(v)$. Here, we use the fact that the interpolation points of f have monotonically increasing resp. decreasing travel time and energy consumption values (Chapter 3).

We set the key of a vertex in the priority queue as $\min_\tau^{\ell_i}$, where ℓ_i is the first unprocessed label (i.e. i is minimal) in $\mathcal{B}(v)$. The bag $\mathcal{B}(v)$ must contain at least one unprocessed label, as only vertices with unprocessed labels are kept in the queue. Due to the positive travel time costs in G, the priority queue keys are now monotonically increasing.

Note that in each iteration, we process a label from $\mathcal{B}(v)$ where v is at the head of the queue. If $\mathcal{B}(v)$ has more unprocessed labels, the key of v changes and we must update the position of v in the queue.

Merge Operation and Domination Test. Consider a bag $\mathcal{B}(u) = \{\ell_1^u, \ldots, \ell_n^u\}$ where $\ell := \ell_o^u$ is the first unprocessed label. When we link ℓ with an edge $e = (u, v)$, the operation is obviously limited to ℓ. However, when we merge h_e^ℓ into $\mathcal{B}(v) := \{\ell_m^v, \ldots, \ell_m^v\}$, it is possible that multiple labels in $\mathcal{B}(v)$ are affected. Namely, the labels $\mathcal{L} \subseteq \mathcal{B}(v)$ with a travel time interval which intersects $[\min_\tau^\ell, \max_\tau^\ell]$ (the travel time interval of ℓ). To determine the contents of $\mathcal{L}$ we have two options. We choose to do a linear scan of $\mathcal{B}(v)$. As the labels of $\mathcal{B}(v)$ are sorted by minimum travel time, an alternative is doing a binary search. Here, we choose the first option.

Let $\mathcal{L} = \{\ell_i^v, \ldots, \ell_j^v\}, 1 \leq i \leq j \leq m$. To merge h_e^ℓ into $\mathcal{L}$ with Algorithm 3.1, we assemble the function f represented by $\mathcal{L}$ by "gluing" the labels in $\mathcal{L}$: between two labels ℓ_k^v and ℓ_{k+1}^v we insert an interpolation point $(\min_C^{\ell_k^v}, \min_\tau^{\ell_{k+1}^v})$. We then apply the merge algorithm, computing $\ell_i \cup f$. A linear sweep splits $\ell_i \cup f$ into convex sub-functions $\mathcal{L}' := \{(h_e^\ell \cup f)_1, \ldots, (h_e^\ell \cup f)_k\}$ and we replace $\mathcal{L}$ with $\mathcal{L}'$ in $\mathcal{B}(v)$. Last, we set the first unprocessed index to the minimum of the previous index and the first label changed by the merge. Note that the merge operation retains the ordering in $\mathcal{B}(v)$ and so no sorting is required at any point of the algorithm: the ordering is held as an invariant.

An alternative merge procedure splits h_e^ℓ into portions which match the travel time intervals of the labels in $\mathcal{L}$. As h_e^ℓ is convex, so are the portions which result from this split. We may therefore sweep $\mathcal{L}$, merging each of its labels with the respective portion of h_e^ℓ. This ensures that we merge only convex sub-functions. If the merging of convex functions has advantages over the arbitrary merging, we may exploit these here. However, we are unaware of such advantages and our merging routine does not benefit from convex merging. We therefore do not pursue this alternative.

The approach also ensures incident labels $\ell_j, \ell_{j+1} \in \mathcal{B}(v)$ have a common point, namely $p := (\min_\tau^{\ell_{j+1}}, \min_C^{\ell_j})$. While p is also present in f_v of the profile algorithm, $\mathcal{B}(v)$ obviously contains p twice. Further, p is only necessary if ℓ_j and ℓ_{j+1} intersect within an explicit segment of either label (Figure 4.1 on the next page). If the intersection is outside the travel time intervals of both labels, then we do not need to store p. We only need to insert p between ℓ_j and ℓ_{j+1} during the assembly of f, as well as remove p after the merge if still

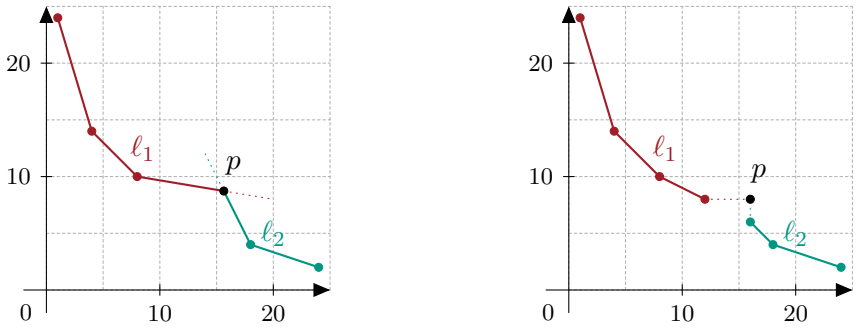


Figure 4.1.: An intersection between labels ℓ_1 and ℓ_2 inside one of their travel time intervals (left) resp. outside both intervals (right). In the latter case, the point p is implicitly given by the last point of ℓ_1 and the first point of ℓ_2.

present. In this manner we reduce some of the redundancy introduced by the label setting algorithm.

Due to the bag order and our choice in priority queue keys, we may not need to consider all of $\mathcal{B}(v)$ when we determine $\mathcal{L}$. Let $\ell' := \ell_o^v$ be the first unprocessed label of $\mathcal{B}(v)$ before we merge h_e^ℓ into $\mathcal{B}(v)$. In the current iteration of the algorithm we poll ℓ from $\mathcal{B}(u)$. Therefore, either $o = 1$, or the minimum travel time of ℓ_{o-1}^v is strictly less than that of ℓ: if ℓ' is not the first label in $\mathcal{B}(v)$, then the previous label in $\mathcal{B}(v)$ was processed in a previous iteration. Therefore, in the ordering of $\mathcal{B}(v)$, we know that ℓ may lie at most between labels o and $o-1$. Linking with e will increase the minimum travel time of ℓ, and so h_e^ℓ also lies strictly after (w.r.t. τ axis) the first point of ℓ_{o-1}. Our linear scan or binary search may thus start from $o-1$. Figure 4.2 on the following page illustrates this observation.

The domination test functions in the same way. We first determine $\mathcal{L}$, as during the merge. Then we test whether the function represented by $\mathcal{L}$ dominates ℓ with Algorithm 3.5. As the domination tests does not change $\mathcal{B}(v)$, the routine requires no further work.

Target Pruning and Timestamps. As in the label correcting algorithm, we apply target pruning. In each iteration we test whether the target bag $\mathcal{B}(t)$ dominates the current label $\ell \in \mathcal{B}(v)$. For this too, we use the adjusted domination routine. If $\mathcal{B}(t) \propto \ell$ holds, we can prune ℓ.

The time-stamping technique sets an empty bag $\mathcal{B}(v) = \emptyset$ instead of an empty function, whenever an edge (u, v) to a vertex v with an outdated stamp is relaxed.

Correctness. The pseudocode of the label setting approach can be seen in Algorithm 4.1 on page 41. We now show that the algorithm is label setting. We then prove the algorithms correctness, i.e. the approach computes the same function as the standard profile algorithm. Let s and t be the source resp. target vertices of a profile query. Further let f_t be the profile computed by the profile search purposed in Section 4.1.

Theorem 4.1. *Algorithm 4.1 is label setting.*

Proof. The label setting property is satisfied by the algorithm iff each label is processed at most once. In order to process a label more than once, a merge operation needs introduce an improvement in the labels tradeoff function. Furthermore any change in the unprocessed labels of a bag, during a merge, is of no importance: these labels are not yet propagated by the algorithm.

Consider a bag $\mathcal{B}(v) = \{\ell_1, \ldots, \ell_n\}$ with last processed label ℓ_i. A merge $h \cup \mathcal{B}(v)$ may at most change ℓ_i due to our earlier observations. More specifically, the change may not occur

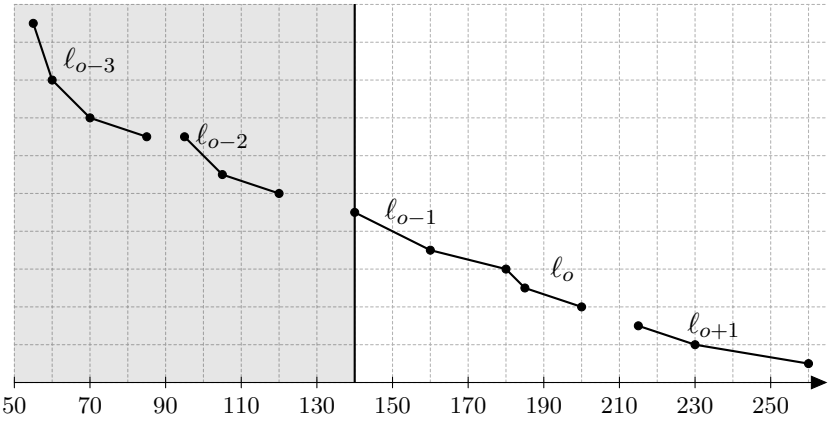


Figure 4.2.: A section of $\mathcal{B}(v)$ which contains the first unprocessed label at the vertex v. The previous label ℓ_{o-1} is therefore already processed and so the priority queue minimum key must be greater than the minimum travel time of ℓ_{o-1}. No unprocessed label at any vertex u may therefore start in the grey area. As a result, only labels with indices higher or equal than $o-1$ are important for the merge operation and the domination test.

in the interval $[-\infty, \min_\tau^{\ell_i}]$. Merging ℓ_i with h will at most cut ℓ_i at $\min_\tau^h$, introduce new labels with indices greater than i to $\mathcal{B}(v)$ and shift the labels in $\mathcal{B}(v)$ with indices greater than i to the right. The processed labels of $\mathcal{B}(v)$ will therefore not be propagated a second time. □

Theorem 4.2. *Upon termination of Algorithm 4.1, the function of $\mathcal{B}(t)$ equals f_t.*

Proof. As seen in the previous proof, the algorithm propagates any change due to a merge while maintaining the label setting property. Furthermore, the algorithm may terminate only if there are no more labels left to propagate, i.e. all improvements are propagated.

Similar to the standard profile algorithm, as long as the edge weights are positive (both consumption and travel time) target pruning does not hinder the correctness of the approach. □

Path extraction. In order to extract actual paths, we keep an additional parent field ℓ_p in each label $\ell \in \mathcal{B}(v)$. Given sensible parent information, we may restore the path from which a given label is constructed by retracing the parents of the label.

During a link operation $\ell \circ c(e)$, $e = (u, v)$, we set the parent of the resulting label $\ell' \in \mathcal{B}(v)$ to ${\ell'}_p := u$. A merge operation, on the other hand, does not change the parent of a label: at most we cut dominated intervals from the function stored in the label.

We then reconstruct the path of a label $\ell \in \mathcal{B}(t)$ as follows. We find the "parent label" of ℓ, by linking each label $\ell' \in \mathcal{B}(v), v = \ell_p$ with $c(e), e = (v, t) \in E$. The parent label of ℓ is then the resulting $\ell' \circ c(e)$ which dominates ℓ. This we repeat until we reach the source vertex s, resulting in the desired path.

The intuition behind determining each parent label is simple. At a vertex v we may have multiple labels $\ell' \in \mathcal{B}(v)$. Each such label results from two operations. First a link with an edge (u, v), and then merging the result of this link operation with the labels already in $\mathcal{B}(v)$. Thus, linking the parent label of ℓ' results in a label that is either equal to ℓ', or spans a wider travel time interval which is then cut by other labels in $\mathcal{B}(v)$.

Algorithm 4.1: MLCProfile

Input: Graph $G = (V, E, c)$, source vertex s, target vertex t
Data: Priority queue Q
Output: Convex sub-function set $\mathcal{B}(t)$ representing profile function at t w.r.t. c.

```
1  B(s) ← {(0, 0)}
2  Q.UPDATE(s, 0)
3  while Q is not empty do
4      u ← Q.PEEKMIN()
       // "Pop" first unprocessed
5      ℓ ← B(u).FIRSTUNSETTLED()
6      if ¬B(u).HASUNSETTLED() then
7          Q.DELETEMIN ()
8      else
9          Q.UPDATE(u, B(u).KEY())
       // Target pruning, sweep unprocessed portion of B(t)
10     if B(t).DOMINATES(ℓ) then
11         continue
12     forall (u, v) ∈ E do
           // Convex labels allow convex linking
13         h_e^ℓ ← LINKCONVEX(ℓ, c(e))
           // Sweep unprocessed portion of B(v), determine L
14         if ¬B(v).DOMINATES(h_e^ℓ) then
               // Merge h_e^ℓ with L, update first unprocessed
15             B(v).MERGE(h_e^ℓ)
16             Q.UPDATE(v, B(v).KEY())
```

Speed directions. Having extracted the path P of a label $\ell \in \mathcal{B}(t)$, we now pick a point p on the convex sub-function ℓ. We wish to know the travel time spent on each edge along the path, from which p originates. By using this information we can derive driving speed directions, which allow reaching t with the consumption value of p.

Consider the last edge $e = (v, t)$ on P and let $\ell' \in \mathcal{B}(v)$ be the parent label of ℓ. Therefore, the point p lies on the link $\ell' \circ c(e)$ and so p originates from the sum of a point p_a on ℓ' and a point p_b on $c(e)$. I.e. $p = p_a + p_b$, and computing p_b yields the desired travel time spent on e.

We compute the point p_b as follows. On ℓ, p lies on a particular segment s. According to our observations in the convex linking case (Section 3.2), s is either of the form $\overline{a_i b_j a_{i+1} b_j}$, or $\overline{a_i b_j a_i b_{j+1}}$. In the first case any point on s is the sum of b_j and the segment $a_i a_{i+1}$ on ℓ', i.e. $p_b = b_j$. In the second case, p results from the sum of a_i and a point on $\overline{b_j b_{j+1}}$, i.e. $p_b = p - a_i$.

Once we know the travel time spent on e, we can continue with the second to last edge e' on P, the label ℓ' and the parent label of ℓ'. We compute the travel time spent on e' in the same manner, then continue with the edge on P that is previous to e'. By the time we reach the edge on P that is incident to the source vertex s, the travel time on each edge on P is known.

5. Advanced Techniques

Having designed the basic algorithms which compute travel time and consumption profiles, we now explore speed-up techniques common in route planning. More specifically, we examine the application of the A* search and the adaptation of the CH algorithm. Similar to the previous chapter, $G = (E, V)$ is a weighted directed graph, the edge costs are tradeoff functions and the travel times of all interpolation points are positive.

5.1. A* Search

We eliminate the negative consumption costs on edges by applying the A* algorithm in the same way as Baum et al. in [BDHS+14]. To gain non-negative reduced edge costs during an s–t query we require lower bounds on $\mathrm{d}(v, t), v \in V$ (Section 2.4). Furthermore, we may improve the running times of both algorithms introduced in Chapter 4 with the goal direction of A*. In our case we operate with the travel time and the energy consumption distances, and so we may use lower bounds on both to better guide the search.

To obtain the travel time potentials $\pi_\tau(v), v \in V$, we run a backward Dijkstra using minimal travel time as a single metric, i.e. $\min_\tau$ of $c(e), e \in E$. We do not apply a stopping criterion. The edge costs are positive and so the algorithm is vertex setting. As we use $\min_\tau$ at all edges, the resulting distances are lower bounds for any travel time we may compute during the profile algorithms.

We compute the consumption potentials $\pi_{\mathrm{C}}(v), v \in V$ in the same manner, respectively we use the minimum consumption $\min_{\mathrm{C}}$ of $c(e), e \in E$ as a single metric. We ignore battery constraints and so the edge costs are no longer positive and the algorithm is not vertex setting. In other words, computing the consumption vertex potentials involves more computation than the travel time counterpart. Despite this, the time we need to compute the consumption potentials is dominated by the running times of the profile algorithms (Chapter 6). Once again, note that negative consumption cycles are physically impossible, i.e. Dijkstra's algorithm terminates and computes the correct distances w.r.t. $\min_{\mathrm{C}}$.

Note that we ignore the overcharging constraint in both searches, as we wish to compute lower bounds. We may however apply the undercharging constraint to limit the searches according to M, i.e. whenever the lower bound on consumption exceeds M we prune.

Having computed the vertex potentials, we may now use A* goal direction. Instead of actually shifting the function f_v at a vertex v by the potential $\pi(v) = (\tau, \mathrm{C})$, we only

transpose the priority queue key of f_v. We do not shift functions, as we require the unchanged consumptions in order to apply battery constraints. Changing the order in which we process vertices (labels in the case of Algorithm 4.1) is enough to gain the speed-up of A*. The changed processing order also ensures the correctness of target pruning, as we implicitly operate on non-negative reduced edge costs [BDHS+14].

Interval Pruning. Having computed the fastest path P_τ to obtain the travel time potential, we may use the consumption C of P_τ as an upper bound on energy consumption. Any s–t path with travel time τ' greater than that of P_τ and optimal consumption (w.r.t. τ') will require at most C consumption. Thus, any function f_v with $\min_{\mathrm{C}}^{f_v} > \mathrm{C}$ is infeasible and may be safely pruned. Moreover, we prune if $\pi_{\mathrm{C}}(v) + \min_{\mathrm{C}}^{f_v} > \mathrm{C}$ as f_v will be shifted at least by $\pi(v)$ before reaching t.

We may use an upper bound $\max_\tau$ on travel time in the same manner, however obtaining the bound requires more work. We now run a forward search from s, which obeys the battery constraints and uses $\min_{\mathrm{C}}^{c(e)}, e \in E$. I.e. we compute the consumption optimal path P_{C}. We set $\max_\tau$ to the travel time of P_{C}: any less energy efficient path will require less travel time to traverse.

Note that in the case of the label setting algorithm we prune the current label of the iteration, i.e. the first unprocessed convex sub-function of f_v and not the whole function f_v.

Improved Target Pruning. Using the vertex potentials we may also improve the target pruning of the two algorithms introduced in Chapter 4. Namely, we may prune f_v as soon as f_v transposed by $\pi(v)$ (denoted by $f_v + \pi(v)$) is dominated by f_t. Here we use the fact that any v–t path must cost at least $\pi(v)$, as the vertex potentials offer lower bounds on travel time and consumption. The minimum profile of reaching t by using f_v is therefore $f_v + \pi(v)$.

5.2. Contraction Hierarchies

To speed-up the profile queries further we adapt the time-dependent variant of the Contraction Hierarchies algorithm (TDCH, [BGNS10]). There are two obstacles which hinder the direct application of TDCH in our context. First, in electric vehicle routing we must obey battery constraints which are not present in the time-dependent scenario. And second, the properties of our profile functions differ from the ones used by Batz et al. We deal with the first problem during the offline phase of CH, by utilizing upper and lower bounds. We then use the approximated TDCH search of Batz et al. (ATCH) during the query phase. Due to the second issue we must also adapt the interval search phase of the ATCH.

We now explain the offline preprocessing phase of our adaptation in detail, that is the contraction of the graph G. We then proceed with the changes required in the query phase, which uses the contracted graph to speed up profile queries. To avoid redundancy we assume the reader to be familiar with the concepts of TDCH, [BGNS10].

5.2.1. Offline Phase

As outlined in the preliminaries section, during the preprocessing phase of CH we contract the vertices of G and add shortcuts to G as needed. Let $R = (V, E_R)$ denote the overlay graph which results from vertex contractions (see [BGNS10]). Similar to Batz et al. we compute the contraction order online with the help of simulated contractions. For each $v \in V$ we compute the impact on R, if v is contracted. The vertex contraction cost used by Batz et al. is as follows:

$$\mathrm{cost}\, v := 2 \cdot \mathrm{edges}\, v + \mathrm{unpacked}\, v + \mathrm{depth}\, v + 2 \cdot \mathrm{complexity}\, v$$

For the definitions of edges, unpacked, depth and complexity see [BGNS10]. We separate each portion of the cost into *removed* and *inserted* cost, e.g.:

$$\text{edges}\, v := \frac{\#\text{ inserted shortcuts}}{\max(1, \#\text{ removed shortcuts or edges from } R)}$$

In the addend edges v of the cost we choose to always count a shortcut $s = (u, w)$ in the inserted shortcuts count. If $e = (u, w) \in R$ then we increase the removed portion of the edges v cost. Furthermore, if the shortcut s will replace the edge e we count the cost of e as removed cost and the cost of s as inserted cost. For instance if the s shortcut function is merged into the function of e, the inserted complexity v cost caused by (u, w) equals the complexity of the shortcut function. The removed portion of the complexity v cost, in turn, is increased by the complexity of e.

At the beginning of the preprocessing we simulate the contraction of each vertex, which we refer to as the initial simulated contractions. We then contract the vertices of G according to their contraction cost in ascending order. When we contract $v \in R$, the cost of all the neighbours of v in R changes. Each neighbour loses an edge (or a shortcut) to v and possibly gains new neighbours. We must therefore simulate the contraction of the neighbours of v anew, updating their cost.

To ensure shortest path preservation *in the time-dependent scenario*, we examine all pairs of neighbours (u, w) of v with $e_1 = (u, v), e_2 = (v, w) \in E_R$. We then test whether the linked cost $h := c(e_1) \circ c(e_2)$ of the path $u \to v \to w$ is dominated by the u–w profile in $R \setminus \{v\}$. If so, a witness is found and we do not require a shortcut from u to w in R: v does not lie on any shortest u–w path. Otherwise, we add the shortcut (u, w) to R with cost h. If $e = (u, w) \in E_R$ we must merge h into $c(e)$.

Due to battery constraints, we cannot use this approach to compute our contraction hierarchy. As explained in Chapter 4, the profile algorithms handle battery constraints by cutting profiles below 0 and above M after each link operation. This requires knowledge of the current battery capacity, which we lack during the CH preprocessing: a shortcut must be usable for all SoC values. Thus, a witness must be found for every battery capacity in order to avoid a shortcut. Furthermore a shortcut (u, w) represents multiple paths, some of which may not be feasible for a specific SoC. A possible way of encoding this information is to include the state of charge as a parameter to the cost functions. However, we wish to avoid three dimensional functions and so use bounds instead.

Shortcut Bounds. Consider the link $\underline{\ell} := f \circ g$, with $e_1 = (u, v), e_2 = (v, w) \in E$ and $f = c(e_1), g = c(e_2)$. If we do not cut consumption values of $\underline{\ell}$ outside $[0, \mathrm{M}]$ then $\underline{\ell}$ is a lower bound for the profile h on $u \to v \to w$. As we allow the full recuperation on e_1 and e_2, for any given SoC the consumption of h will be at least that of h for all τ.

We may also forbid recuperation on $u \to v \to w$ by cutting both f and g at 0, resulting in $\overline{f}$ and $\overline{g}$. Computing the linked function $\overline{\ell} := \overline{f} \circ \overline{g}$ yields an upper bound for h. The less recuperation on $u \to v \to w$ is possible (due to overcharging), the higher the consumption of h is. As we allow no recuperation at all when we compute $\overline{\ell}$, no consumption of h may lie above $\overline{\ell}$.

Before the initial simulated contractions we set a lower bound $\underline{e}$ and an upper bound $\overline{e}$ for each edge $e \in E$. Namely, $\underline{e} = c(e)$ and $\overline{e} = c(e)_{\geq 0}$. As we just observed, linking the lower resp. upper bounds of the two edges $e_1 = (u, v)$ and $e_2 = (v, w)$ results in valid lower and upper bounds for the profile h. If an edge $e' = (u, w)$ exists, we may merge $\underline{e_1} \circ \underline{e_2}$ into $\underline{e'}$ and $\overline{e_1} \circ \overline{e_2}$ into $\overline{e'}$ thus bounding both h and $c(e')$. Observe that whether e_1, e_2 and e' are edges or shortcuts is not relevant. As long as we link and merge the respective bounds, the resulting new lower and upper bounds remain correct.

Having defined the shortcut bounds, we now have all ingredients we require to compute our contraction hierarchy. Whenever we contract a vertex $v \in R$ we examine all neighbour pairs of v in R. For each such pair (u, w) with $e_1 = (u, v), e_2 = (v, w) \in E$ we compute $\underline{h} := \underline{e_1} \circ \underline{e_2}$ and $\overline{h} := \overline{e_1} \circ \overline{e_2}$. We then compute the u–w upper bound profile $\overline{g}$ in $R \setminus \{v\}$ by using solely upper bounds of edges and shortcuts in R. If $\overline{g}$ dominates $\underline{h}$, then any u–w profile through v is dominated by u–w paths that do not pass through v. In such case we do not require a shortcut, we remove v from R and proceed with the next contraction. Otherwise we must add a shortcut (u, w) to R prior to removing v, as a path might pass through v which is important for the u–w profile at some SoC.

Whenever we add a (u, w) shortcut, we first check whether an edge or a shortcut from u to w exists in R. If not, we add (u, w) to the set of edges of R, with bounds $\underline{h}$ and $\overline{h}$. Furthermore, we initialize the edge set of (u, w) to $S := \{(u, v), (v, w)\}$. This set contains the subgraph represented by the shortcut and is empty for all original edges of G. As before, let $\underline{h}$ and $\overline{h}$ be the bounds of (u, w). If $e = (u, w) \in E$ we check whether $\overline{h} \propto \underline{e}$. In such case we know that all paths in the original edge or shortcut from u to w are dominated by the paths of the new (u, w) shortcut. We therefore set the bounds of e to $\underline{h}$ and $\overline{h}$, and the edge set to S. If $\overline{h} \not\propto \underline{e}$, then we merge $\underline{h}$ into $\underline{e}$ and $\overline{h}$ into $\overline{e}$, and add S to the edge set of e.

In order to keep tight bounds we introduce the gap portion of the contraction cost. For each edge or shortcut e we keep track of the maximum consumption difference between the lower bound $\underline{e}$ and the upper bound $\overline{e}$, which we denote by the *recuperation gap* (or simply gap) between $\underline{e}$ and $\overline{e}$. To compute the recuperation gap between an upper and a lower bound it is enough to modify the domination test, Algorithm 3.5. We sweep both bounds in the same manner and we maintain a maximum vertical distance. Instead of reporting whether a point (τ, C) of one bound is above or below a segment of the other bound, we evaluate the segment at τ and update the maximum distance if necessary. That the result is the recuperation gap follows from similar observations to the ones in Theorem 3.11. Thus we define the gap cost of v as:

$$\operatorname{gap} v := \frac{\sum_{e \in \text{inserted shortcuts}} \operatorname{gap} e}{\max(1, \sum_{e \in \text{removed shortcuts or edges from } R} \operatorname{gap} e)}$$

Having chosen the recuperation gap cost, we redefine the vertex contraction cost of Batz et al. as follows:

$$\operatorname{cost} v := 2 \cdot \operatorname{edges} v + \operatorname{unpacked} v + \operatorname{depth} v + 2 \cdot \operatorname{complexity} v + \operatorname{gap} v$$

We must also adjust the complexity portion of the vertex contraction cost, as we no longer have a single function per edge or shortcut. We sum the interpolation points of both lower and upper bounds of an edge or shortcut, and use the resulting count in the complexity cost formula of Batz et al.

Profile and Interval Witness Searches. As in the publication of Batz et al., we utilize additional information in order to avoid expensive profile witness searches or to reduce their cost. Due to the nature of our bounds, as soon as the lower bound $\underline{f}$ of a path $u \to v \to w$ allows negative consumption we cannot find a witness: the upper bounds are strictly positive. And, of course, as soon as a witness is found we stop the witness search.

Furthermore, we take advantage of interval searches. Again, we wish to know if a witness exists for a path $u \to v \to w$. Consider the lower and upper bounds of this path, $\underline{\ell}$ resp. $\overline{\ell}$. Also consider the lower and upper bound profiles we may compute in R, $\underline{f}$ resp. $\overline{f}$. As before, let $\min_\tau$,$\min_\mathrm{C}$,$\max_\tau$ and $\max_\mathrm{C}$ denote minimum and maximum values of travel

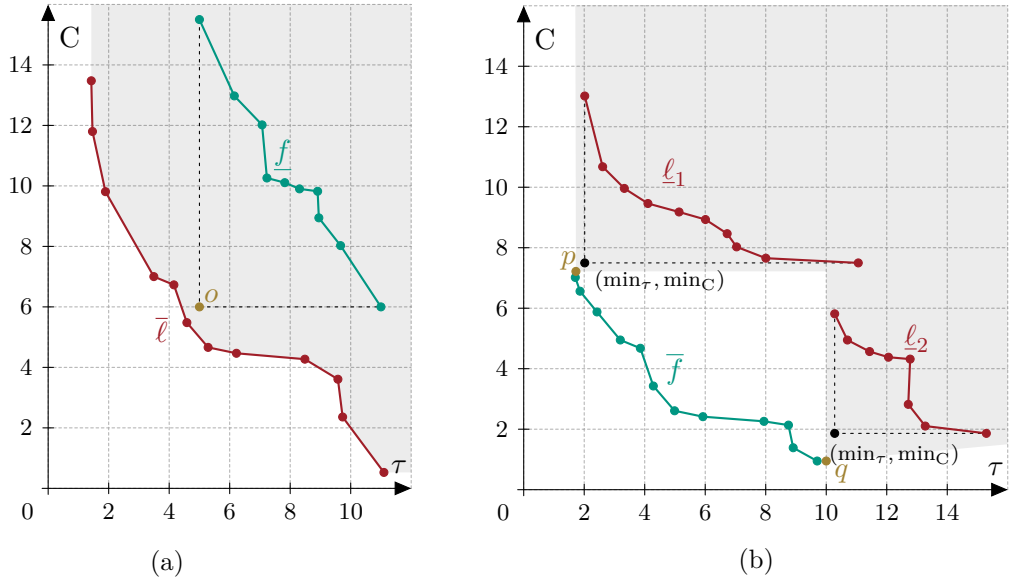


Figure 5.1.: Witness information based on interval searches. We may compute the minimum travel time and consumption of the lower bounds (left) resp. upper bounds (right) of u–w paths in R. In the first case, if the resulting tradeoff point o is dominated by the $u \to v \to w$ link upper bound $\overline{\ell}$, then no witness may be found. And in the second case, the resulting tradeoff points p and q may dominate the shortcut lower bound, e.g. $\underline{\ell}_1$ or $\underline{\ell}_2$. If so, then a witness exists. The grey area in both figures represents points dominated by $\overline{\ell}$, resp. p and q.

time and consumption. If the $(\min_\tau^{\underline{\ell}}, \min_C^{\underline{\ell}})$ point of $\underline{\ell}$ is dominated by $\overline{f}$ then a witness exists. On the other hand, if $\overline{\ell}$ dominates the $(\min_\tau^{\underline{f}}, \min_C^{\underline{f}})$ point of $\underline{f}$, then no witness can be found. We therefore utilize two interval searches which use this information.

First we compute the upper bound minimum travel time and consumption u–w paths in R, denoted by $\overline{P_\tau}$ resp. $\overline{P_C}$. For this we use the standard Dijkstra algorithm with scalar edge weights. The travel times and consumptions of $\overline{P_\tau}$ and $\overline{P_C}$ are tradeoff points which we denote by p resp. q. The $(\max_\tau^{\overline{f}}, \min_C^{\overline{f}})$ point of $\overline{f}$ dominates q. To clarify, we compute $\overline{P_C}$ as *any* energy-optimal path. The consumption of $\overline{P_C}$ is therefore equal to $\min_C^{\overline{f}}$, the travel time however may be higher than $\max_\tau^{\overline{f}}$. Similarly, the (minimum travel, time maximum consumption) point $(\min_\tau^{\overline{f}}, \max_C^{\overline{f}})$ dominates p. Thus, if either p or q dominates $o := (\min_\tau^{\underline{\ell}}, \min_C^{\underline{\ell}})$ then we know that o must also be dominated by $\overline{f}$. In other words, we know that a witness exists. Figure 5.1b illustrates this observation. We denote this search by the *upper bound interval search*.

Second we compute lower bound minimum travel time and consumption u–w paths in R, denoted by $\underline{P_\tau}$ resp. $\underline{P_C}$. Here we may also use Dijkstra with scalar edge weights. However we require reduced edge costs when computing $\underline{P_C}$, as lower bounds allow negative consumption. Thus we use height induced vertex potentials (Section 2.4 on page 13), which we compute once for G before any contraction. As the contractions do not change shortest path distances, the potentials remain correct throughout the preprocessing. The computed minimum values $\min_\tau^{\underline{f}}$ and $\min_C^{\underline{f}}$ form a point $o := (\min_\tau^{\underline{f}}, \min_C^{\underline{f}})$. We then test if $\overline{\ell}$ dominates o and if so, we know no witness can be found. See Figure 5.1a. This search we denote by *lower bound interval search*.

Hop Limit. Similar to Batz et al. we use a hop limit during the interval and profile witness searches. Whenever we look for a witness for the path $u \to v \to w$, we first run

a breath-first search from u and mark any vertex within 16 hops. If we cannot reach w from u within this hop limit we insert a shortcut. Otherwise, we limit allow the interval and profile searches to visit only marked vertices. We choose to mark vertices beforehand and not count hops during each search, in order to speed up profile searches with vertex potentials.

Vertex Potentials. During the upper bound interval search we compute the $\min_{\tau}^{\overline{f}}$ and $\min_{\mathrm{C}}^{\overline{f}}$ of $\overline{f}$, where $\overline{f}$ is the u–w upper bound profile in $R \setminus \{v\}$. Moreover, we compute a minimum travel time and consumption value from u to each vertex v within the hop limit. Due to the triangle property of shortest paths, we may use the computed values as vertex potentials: $\pi_\tau(v) := \mathrm{d}_\tau(u, w) - \mathrm{d}_\tau(u, v)$ and $\pi_{\mathrm{C}}(v) := \mathrm{d}_{\mathrm{C}}(u, w) - \mathrm{d}_{\mathrm{C}}(u, v)$.

In this manner we obtain valid lower bounds on the v–w travel time and consumption distance, however we can do better. Namely, we may run the upper bound interval search *backwards* from w to u. In this case we know the exact minimum travel time or consumption distances from v to w. These distances offer the optimal vertex potentials we may use during a profile search, so that the profile search remains correct.

Running a backward search from w will generally visit different nodes compared to a forward search from u, both within the given hop limit. For this reason we choose to first mark the vertices reachable from u and then run interval and profile searches.

Target Pruning. During the witness search we use $\underline{\ell}$ for target pruning. Whenever a vertex v is removed from the queue and if the profile at v is dominated by $\underline{\ell}$, then we do not continue the search from v. In this case the profile at v has no improvement over $\underline{\ell}$ and so is unimportant. We also use the standard target pruning during a witness search, i.e. when the profile at v is dominated by the current profile at w (the target vertex) we prune v. Note that upper bounds in R are non-negative and so both types of pruning do not require any vertex potentials.

We apply similar pruning in both interval searches. The standard target pruning functions in the same manner as in the algorithm of Dijkstra, as the interval searches utilize scalar shortest paths. Using ℓ to limit the interval searches, however, is less straightforward: in both the upper and the lower bound interval searches we assume that w is reachable from u in $R \setminus \{v\}$ within the given hop limit. As before let f and ℓ denote the u–w profile in $R \setminus \{v\}$ resp. the $u \to v \to w$ link.

When we compute the upper bound minimum consumption in R (during an upper bound u–w interval search) we prune vertices with consumption higher than $\min_{\mathrm{C}}^{\underline{\ell}}$. If we do not reach w, the minimum consumption of $\overline{f}$ is above $\min_{\mathrm{C}}^{\underline{\ell}}$ of $\underline{\ell}$ and $\overline{f}$ cannot dominate $\underline{\ell}$. Similarly, during the upper bounds minimum travel time computation of the same search, we prune vertices with travel time higher than $\min_{\tau}^{\underline{\ell}}$. Again, if w is not reached, $\overline{f}$ cannot dominate $\underline{\ell}$. Figure 5.2b on the facing page illustrates the two cases where pruning hinders reaching w.

Due to this pruning, some vertices reachable from u within the hop limit will not be visited by the upper bound interval search. If we wish to apply the computed values as vertex potentials during the profile witness search, it is most beneficial if all vertices are visited and have values. To achieve this, we must disable the pruning. We thus choose a specific percentage of contracted vertices as a threshold (Chapter 6). When R has less vertices than this threshold, we disable the upper bound interval search pruning. The resulting interval searches become slower to the benefit of faster profile searches.

During the lower bound u–w interval search we limit both scalar searches with the $(\max_{\tau}^{\overline{\ell}}, \max_{\mathrm{C}}^{\overline{\ell}})$ point of $\overline{\ell}$. In other words, we prune vertices with higher values as we do in the upper bound interval search. If either the minimum travel time or consumption

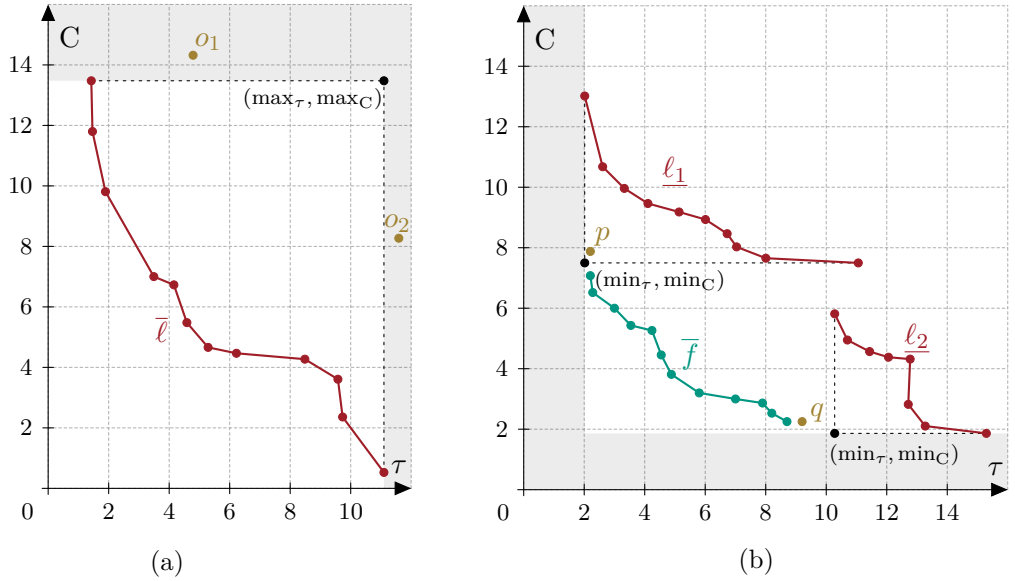


Figure 5.2.: Target pruning based on link profile during lower bound (left) resp. upper bound (right) interval search. In the lower bound case the consumption of o_1 and the travel time of o_2 are too high, which results in unreachable target vertex during the respective scalar search. We clip o_1 to $\max_C$ and o_2 to $\max_\tau$ and then test if $\overline{\ell}$ dominates them. In the example to the right we do not reach p when conducting upper bound interval search for ℓ_1, as the minimum travel time of $\overline{f}$ is too high. A similar situation occurs with q and ℓ_2 but due to minimum energy consumption.

search does not reach w, we set $\max_\tau^{\overline{\ell}}$ resp. $\max_C^{\overline{\ell}}$ as sentinel values: not reaching w means the respective value of $\underline{f}$ is higher than the limit. Thus, if $\overline{\ell}$ dominates the "sentinel point", then $\overline{\ell}$ also dominates the actual $(\min_\tau^{\underline{f}}, \min_C^{\underline{f}})$ point of $\underline{f}$. See Figure 5.2a. Due to the height-induced potentials we use during the lower bound interval search, negative edge consumption costs are no obstacle to the pruning of either search.

Whenever we choose to use vertex potentials in the profile witness search we may also apply the improved target pruning, used in the basic profile algorithms. We prune v if the profile at v, transposed by the potentials at v, is dominated by either $\underline{\ell}$ or the current profile at w.

Tighter Bounds. So far, to obtain the $u \to v \to w$ bounds we link the lower resp. upper bounds of edges or shortcuts (u, v) and (v, w) in R. This results in bounds which are generally further apart after each link. I.e. the further we are in the contraction of G, the wider the gap between the upper and lower bounds of an edge in R is. Of course, wider gaps result in more paths stored in a shortcut as the profiles of more paths "fit" within the shortcut bounds. In turn, this means our query will visit all such paths if the shortcut is required. We therefore wish to inhibit the widening of the bound gaps as much as possible.

Consider an original edge e in G with an upper bound $\overline{g}$ and $\underline{g}$ of $g := c(e)$. We defined the recuperation gap as the maximum energy recuperation allowed by e. Due to our choice in bounds, this gap equals $\max_\tau(\overline{g}(\tau) - \underline{g}(\tau))$ i.e. the maximum vertical distance between the two bounds. The recuperation gap of a shortcut s in R is also equal to the maximum distance between the shortcut bounds, as the upper bound allows no recuperation for a given τ and the lower bound allows full recuperation for τ.

Furthermore, the lower bound $\underline{\ell}$ of a shortcut $s = (v, w)$ consists of paths in G, on which no overcharging occurs. We may thus realize $\underline{\ell}$, if the state of charge SoC is low enough before

traversing s. Formally, if SoC plus the recuperation gap γ of s is not greater than M no overcharging may occur. In this case we know that the profile of s is $\underline{\ell}$. Suppose a further shortcut h has minimum consumption $\min_C \geq \gamma$. This also implies that $M \geq \gamma + \max SoC$ due to $\min_C = M - \max SoC$, where $\max SoC$ is the maximum state of charge we may have after traversing h (staring with a full battery). Thus, when linking h with s we link the lower and upper bounds of h with $\underline{\ell}$: the state of charge after traversing h is in all cases low enough to allow traversing s with $\underline{\ell}$.

Note that each pair of bounds in R has equal minimum travel time, due to our choice of bounds in G, and so the recuperation gap is always finite. As mentioned, the initial recuperation gap of an edge e in G is equal to the difference of minimum consumption between the lower and upper bounds of e. Again, this is not necessarily true for edges (shortcuts) in R due to the merging of shortcut bounds. Thus, each time we add a (u, w) shortcut we compute the recuperation gap of (u, w) anew and store the value.

We may further utilize the idea of linking with the lower bound during interval and profile witness searches. Specifically, we first run the lower bound interval search. This yields minimum consumption from u to each visited vertex x, denoted by $\min C_x$. A vertex y which was not visited due to target pruning has $\min C_y \geq \min_C$, where $\min_C$ is the minimum consumption of the $u \to v \to w$ link. For such a vertex we set $\min C_y := \min_C$. We then run the upper bound interval search, where we "link" edges $s = (x, y) \in R$ to functions at $y \in V$ as the search runs backward. Before such a link, we check whether $\min C_x \geq \gamma_s$ (with γ_s being the recuperation gap of s) or whether s allows no recuperation. If so, we link with the lower bound $\underline{\ell}$ of s. More specifically, during the upper bound travel time interval search we link with the first point of $\underline{\ell}$. During the respective consumption search we link with the last point of $\underline{\ell}$. During a profile witness search we simply link with $\underline{\ell}$ if $\min C_x \geq \gamma_s$ holds during the relaxation of s.

Below a specific contraction percentage we disable the target pruning of the upper bound interval search. We may do the same for the lower bound interval search, in order to obtain better minimum consumption values at vertices. This in turn may allow us to link with lower bounds more often. If we disable target pruning for both searches at a specific percent, one search will asymptotically "cover" the costs of the other search.

Contraction Caching. To avoid redundant computations we apply contraction caching, see [BGNS10]. During the initial contraction simulations, for each pair of neighbours u and w of v we store whether a shortcut is needed for the path $u \to v \to w$. When we contract the first vertex, we know which shortcuts are needed in R due to the cached information. After the contraction we examine the neighbours of v. We first delete cached information which concerns v, i.e. paths $v \to u \to x$ and $x \to u \to v$ where u is a neighbour of v in R and x a neighbour of u. And second, for each added shortcut (u, w) we mark the paths $x \to u \to w$ and $u \to w \to y$. Here x and y are neighbours of u resp. w again in R. When we simulate the contractions of u and w later on, we look for a witness only for the marked paths: whether we need shortcuts between other neighbour pairs or not is answered by the cached information. Any marked path that has no witness is then added to the cache of required shortcuts.

When using contraction caching it becomes necessary to allow only witnesses which dominate the $u \to v \to w$ lower bound strictly. Consider a standard CH preprocessing with a single metric and two paths $u \to x \to w$ resp. $u \to y \to w$ with equal distance. Suppose no other paths from u to w exist. Using standard domination, we find and cache a witness for both paths and contract both y and x without adding an u–w shortcut. In result we lose a shortest path from u to w. As we may construct a similar scenario in our case, we require strictly dominating witnesses.

To fully utilize the contraction caching we may also compute the vertex contraction cost with deltas. As explained previously, we separate the cost of each vertex into removed and inserted cost. The initial simulated contractions set the two costs of each vertex before any contraction. Whenever v loses a neighbour u due to a contraction, we deduct the cost of (u, v) from the removed cost of v. Similarly when we add a shortcut (u, v) we must increase the removed costs of u and v by the cost of the shortcut. Furthermore, during the simulated contraction of v we add the cost of each needed shortcut to the inserted cost of v. If v is actually contracted later on, for each needed shortcut (u, v) or (v, u) we decrease the inserted cost of u. And last, whenever we insert a (u, w) shortcut the inserted cost of any vertex x which also needs a (u, w) shortcut may change. To handle such situations, it is sufficient to look for such a vertex x in the neighbourhoods of u and w in R.

Contracting Degree 1 Vertices. Dead-end vertices in G are a special case during the CH preprocessing. Contracting a vertex of degree 1 introduces no shortcuts and requires no witness searches. Furthermore, dead-ends which are neither s nor t are never important for any s–t query. However, the contraction costs of vertices do not necessarily reflect the importance of contracting dead-ends. The impact of contracting some degree 1 vertex may not be as "positive" as contracting some other vertex with higher degree. We thus contract all degree 1 vertices in a step before the initial simulated contractions.

Parallelization. We may apply the parallelization used by Batz et al. without any adaptations. As this improvement does not yield a contraction hierarchy of higher quality and only improves the preprocessing time, we omit the parallel vertex contractions. However, when simulating the contractions of vertices $\{v_1, \ldots, v_n\}$ we parallelize the required witness searches. The searches are trivial to parallelize and later during the contraction become both numerous and expensive. We therefore apply this parallelization after a specific percentage of vertices has been contracted (see Chapter 6).

5.2.2. Query

Due to the battery constraints, we employ bounds during the CH preprocessing. To answer exact profile queries, we apply the ATCH algorithm of Batz et al. In essence, the query consists of three steps. In the first and second steps we utilize the CH structure and the shortcut bounds to prune provably unimportant edges in G. In the last step we run the standard profile algorithm, using only the remaining edges of G.

We now explain the adaptations we require for the first and second steps in detail. We begin with the first step, an inexpensive interval search, which discards shortcuts based on scalar information. We then proceed with the second step: a more costly bound search. And last, we discuss the use of interval search information to speed up the bound search.

Interval Search

During the interval search phase we wish to discard shortcuts based on scalar information. The structure of the interval search remains as found in [BGNS10], Algorithm 7. Given an s–t query, a forward and a backward search are started from s resp. t. As during a normal CH query, both searches relax edges only to neighbours higher in the hierarchy. The "distance" to a vertex u is an interval $I \subseteq \mathbb{R}$, consisting of the minimum and maximum travel time required to reach u. Relaxing an edge $e = (u, v)$ sums I with the travel time interval of e. We are then able to check whether the upper bound of the interval at v dominates the lower bound of the summed interval. If so, e is not important for the further steps of the query. Furthermore, a tentative upper bound r on travel time is updated whenever a vertex is visited from both searches. Vertices in the common search space, with a lower bound dominated by r, are also not important for the query.

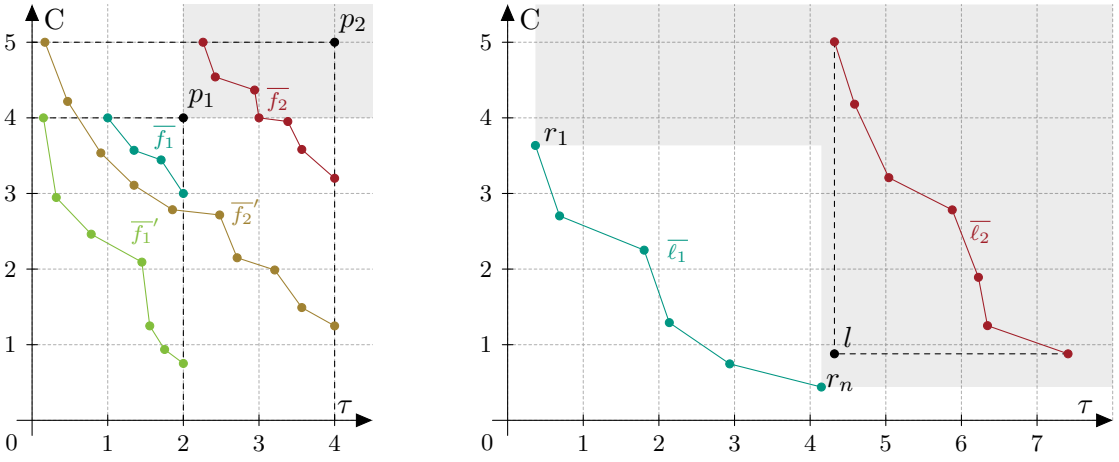


Figure 5.3.: Using a single point for the upper bound during an interval search (left), or using two points (right). The point p_1 may result either from $\overline{f_1}$ or $\overline{f_1}'$. Similarly, p_2 as a valid bound for both $\overline{f_2}$ and $\overline{f_2}'$. Although p_1 dominates p_2, that $\overline{f_1}$ dominates $\overline{f_2}$ is not known. On the other hand, using the extreme points of upper bounds allow some measure of domination checking. If either extreme point r_1 or r_n of one bound $\overline{\ell_1}$ dominates the minimum point l of another bound $\overline{\ell_2}$, then $\overline{\ell_1}$ dominates $\overline{\ell_2}$.

Our search differs, since we have to propagate different types of intervals: an interval in $\mathbb{R}$ is insufficient for our needs. Consider a shortcut s with bounds $\underline{\ell}$ resp. $\overline{\ell}$. The maximum energy consumption of $\underline{\ell}$ and $\overline{\ell}$ for $\tau \in \mathbb{R}$ is ∞. Therefore, unlike the travel time functions of Batz et al., a scalar interval may only constrain $\underline{\ell}$ and $\overline{\ell}$ poorly. Instead we require at least two points, which define a box B (a 2D interval) in $\mathbb{R}$ containing the bounds of s. By extension, B contains any tradeoff function which results from the traversal of s with any given SoC.

For instance, we may choose $(\min_\tau^{\underline{\ell}}, \min_{\mathrm{C}}^{\underline{\ell}})$ of $\underline{\ell}$ and $(\max_\tau^{\overline{\ell}}, \max_{\mathrm{C}}^{\overline{\ell}})$ of $\overline{\ell}$ to define our intervals. We say that $(\max_\tau^{\overline{\ell}}, \max_{\mathrm{C}}^{\overline{\ell}})$ is the *upper* part of some interval and $(\min_\tau^{\underline{\ell}}, \min_{\mathrm{C}}^{\underline{\ell}})$ is the *lower* part. Whenever we link $s = (u, v)$ to the interval $[\underline{\tau}_u, \overline{\tau}_u] \times [\underline{\mathrm{C}}_u, \overline{\mathrm{C}}_u]$ at u we obtain the interval $[\underline{\tau}_u + \min_\tau^{\underline{\ell}}, \overline{\tau}_u + \max_\tau^{\overline{\ell}}] \times [\underline{\mathrm{C}}_u + \min_{\mathrm{C}}^{\underline{\ell}}, \overline{\mathrm{C}}_u + \max_{\mathrm{C}}^{\overline{\ell}}]$. We must then merge this interval into the one present at v, resulting in

$$[\min(\underline{\tau}_u + \min_\tau^{\underline{\ell}}, \underline{\tau}_v), \max(\overline{\tau}_u + \max_\tau^{\overline{\ell}}, \overline{\tau}_v)] \times [\min(\underline{\mathrm{C}}_u + \min_{\mathrm{C}}^{\underline{\ell}}, \underline{\mathrm{C}}_v), \min(\overline{\mathrm{C}}_u + \max_{\mathrm{C}}^{\overline{\ell}}, \overline{\mathrm{C}}_v)].$$

While being valid, the upper parts of such intervals introduce a problem. During the interval search of Batz et al. the upper bound may be improved. I.e. if the upper bound r_v at v is dominated by the upper bound r_u of u linked with s, denoted by r', they set the upper bound at v to r'. Using the maximum travel time and consumption point as an upper bound does not allow such improvements. See Figure 5.3a for an example.

We therefore choose to propagate three points instead of two. Given a shortcut $s = (u, v)$ with upper and lower bounds $\overline{\ell}$ resp. $\underline{\ell}$, we propagate the $o = (\min_\tau^{\underline{\ell}}, \min_{\mathrm{C}}^{\underline{\ell}})$ point of $\underline{\ell}$ as before. For the upper bound $\overline{\ell} = \{r_1, \ldots, r_n\}$ we propagate the two extreme points, r_1 and r_n. Suppose the interval at u is $I_u = \{\underline{o}_u, \overline{p}_u, \overline{q}_u\}$, where $\underline{o}_u$ is the minimum point of the lower bound at u, $\overline{p}_u$ is the first point of the upper bound and $\overline{q}_u$ the last point. To link s to the upper part of I_u we sum the respective points, i.e. $p' = r_1 + \overline{p}_u$ and $q' = r_n + \overline{q}_u$. We wish to merge the resulting upper part into $I_v = \{\underline{o}_v, \overline{p}_v, \overline{q}_v\}$, the interval at v. Let τ be the

travel time of $\overline{p}_v$ and C be the consumption of $\overline{q}_v$. If either p' or q' dominates (τ, C), then the upper bound at v is dominated by that of u linked with s. See Figure 5.3b. In this case we replace the upper bound of I_v with p' and q', which yield a better upper bound at v. Otherwise, we update I_v as follows. If p' offers less travel time than $\overline{p}_v$, then we replace $\overline{p}_v$ with p'. Similarly, if q' offers less consumption than $\overline{q}_v$ we replace $\overline{q}_v$ with q'. In this manner we ensure that the upper part of I_v also contains any function at u linked with s.

We introduce further changes to the interval search of Batz et al. Namely, we alter Stall-on-Demand, the priority queue keys and the common search space pruning. We also apply battery constrains. Last, we use height-induced potentials just as during the interval search of the preprocessing phase.

Stall-on-Demand. While we apply Stall-on-Demand, we do not propagate any stalling. To check whether a vertex u may be stalled during the forward search, we examine all shortcuts $s = (v, u)$ where v is at least as high in the hierarchy as u. If either $r_1 + \overline{p}_v$ or $r_n + \overline{q}_v$ (I_v linked with s) dominates $\underline{o}_u$, then the search at u may be stalled. Stalling during the backward search functions in the same manner, where we examine outgoing shortcuts instead of ingoing ones. Note that in the case of a partial contraction hierarchy, Stall-on-Demand is not applied to the CH core.

Priority Queue Keys. We also require different priority queue keys than the ones used by Batz et al.: we no longer propagate scalar intervals and we traverse shortcuts with possibly negative costs. To amend the latter we use height induced potentials, in the same way we do during the preprocessing phase. As for the keys, we use a combination (Chapter 6) of the travel time and consumption of the $(\min_\tau, \min_\mathrm{C})$ point of our intervals.

Common Search Space Pruning. In the interval search of Batz et al., whenever a vertex v is visited from both searches the algorithm tests for pruning. If the linked lower bound of v is not dominated by the tentative upper bound, pruning is not possible and the vertex is marked for cone unpacking. This is done on the fly, during the searches. Due to the negative consumption of lower bounds in our case, and the bicriteria nature of our searches, we preform this test *after* the interval search terminates. We mark vertices visited by both searches and postpone the pruning test until both the forward and backward searches terminate. Furthermore, we must use strict domination. We may otherwise prune vertices which contribute to the upper bound, and in result to the s–t profile.

Battery Constrains. We also apply battery constraints to some extent. The forward interval search may maintain consumption values in $[0, \mathrm{M}]$ and the backward in $[-\infty, \mathrm{M}]$. Any consumption value results from linking with a shortcut is *clipped* to the respective value range. Both searches may clip to M, as either a forward or a backward path which requires more consumption is infeasible. In the case of a forward path this is self-explanatory. If the consumption of a backward path exceeds M, it must be linked to a forward path with negative consumption in order to be feasible. This however is not possible due to overcharging. For the same reason the forward search may cut intervals at 0. The backward search may not: a path in the backward search with negative consumption becomes feasible as soon as a forward path is linked to it, so that the summed consumption is non-negative.

As for undercharging, whenever the summed (forward and backward) lower bound consumption at some vertex exceeds M, we prune the vertex. In addition, we prune any vertex in the common search space with summed minimum consumption strictly higher than M.

Bound Search

The next phase of the query is the bound search, as found in [BGNS10] Algorithm 12, lines **18** to **31**. A further forward and backward search are started from the source and target vertices. Instead of propagating scalar intervals, the searches propagate travel time

functions. As with the interval search, a lower and an upper bound is maintained at each vertex u. The bounds contain the travel time function to u, and are linked to the respective bounds of $e = (u, v)$ whenever e is relaxed. As during the interval search, if the upper bound present at v dominates the linked lower bound, then the examined path through u is not important for the profile search. And in the same manner a tentative upper bound is maintained and used to prune vertices in the common search space of both searches.

Due to the shape of our tradeoff functions, intervals become unnecessarily large to contain them. For this reason, interval searches often are unable to prune substantial portions of the graph (Chapter 6). It is therefore even more important (than in [BGNS10]) that we conduct bound searches in our scenario: the bounds follow the shape of the tradeoff functions more freely than intervals.

There are a few adaptations and improvements which we require in order to conduct bound searches with feasible running times. Similar to Batz et al. we simplify shortcut bounds after the CH preprocessing. We must also alter the priority queue keys and employ vertex potentials, as shortcut lower bounds allow negative consumption. And last, we take advantage of recuperation gaps to keep the bounds at vertices tight. In the following we explain these adaptations at length.

Battery Constraints and Common Search Space Pruning. As during the interval search, the forward and backward bound searches may clip profile functions to the ranges $[0, \mathrm{M}]$ resp. $[-\infty, \mathrm{M}]$. We also check which vertices in the common search space are important after the bound searches terminate, and not on the fly. Furthermore, the tentative upper bound which we maintain during the bound search is a tradeoff function and not a scalar value. As mentioned before, a scalar value reflects the shape of tradeoff functions poorly – it would thus decrease the effectiveness of the bound search. Similar to the interval search, we employ strict domination: we link the forward and backward lower bound of a common vertex, then test if the tentative upper bound dominates the linked function strictly.

Bound Simplification. Without simplifying the shortcut bounds, running a bound search is very expensive (Chapter 6). As in [BGNS10] we first simplify bounds, before running actual queries. Batz et al. choose the Imai-Iri algorithm [II87] to simplify their travel time functions. However, the Imai-Iri algorithm requires continuous piecewise linear functions. I.e. no two adjacent interpolation points may share an x-axis or y-axis value. This is not the case for our tradeoff functions, as a merge may lead to both equal travel times or consumptions of subsequent points. We may of course choose to simplify the continuous sections of shortcut bounds, however the degree of simplification would depend on the number of discontinuity points.

Instead we choose to apply greedy function simplification, Section 2.6. We lose the approximation guarantee yielded by the Imai-Iri algorithm, however we gain control over the number of interpolation points. During the simplification of shortcut bounds, we obtain lower resp. upper simplified versions of the lower resp. upper original bounds. Furthermore we apply the same function simplification during the bounds search. Whenever we update either the tentative upper bound, or the bounds at some vertex, we check whether the two functions exceed some chosen size. If so, we simplify them.

In this manner we obtain a more detailed "interval" search. We may choose to invest more time in the bound search by allowing more interpolation points for the upper and lower bounds. More detailed functions will allow us to prune more edges. Or, we may conduct a faster search with less accurate bounds. We explore the choice of simplification size in Chapter 6.

Priority Queue Keys and Vertex Potentials. Batz et al. use the minimum travel time of lower bounds as a key during the bound search phase. We also choose to use the lower bound for the priority queue keys. More specifically, we use the same keys as the ones we use during the standard profile algorithm (Section 4.1).

In Chapter 6 we explore query times with different core sizes. Instead of fully contracting the input graph G, we stop the contractions when only a specific percentage of vertices is left uncontracted. Similar to the CALT algorithm [BDS+08] we wish to use potentials during the bounds search in the uncontracted core to speed-up the search. However, we do not use the ALT approach [BDS+08]. Instead, we use the information from the interval search conducted in the previous phase.

During the forward bound search, whenever a core vertex v is pulled from the queue we check whether v was visited by the backward interval search. If so, we use the minimum travel time $\min_\tau$ and consumption $\min_C$ values of the interval search at v to possibly prune v. Or, whenever linking the bounds at u to a shortcut $s = (u, v)$, we use $(\min_\tau, \min_C)$ as vertex potentials when propagating the linked bounds (see Section 5.1).

We prune v if the simplified forward lower bound $\underline{f}$ at v, transposed by $(\min_\tau, \min_C)$, is *strictly* dominated by the tentative upper bound of the interval search. In this case $\underline{f}$ may not lead to an improvement of the s–t profile. We also prune v if $\underline{f} + (\min_\tau, \min_C)$ is dominated by the tentative upper bound of the bounds search. If so, then the current common search space contains a provably better function and the propagation of $\underline{f}$ is not necessary.

If v was not visited by the backward search, or is not a core vertex, we apply height-induced potentials during the linked bounds propagation.

The same potentials and pruning are utilized during the backward search. There, we use the information from the forward interval search.

Tighter Bounds. As during the CH preprocessing, we wish to utilize recuperation gaps as much as possible. During any direction of either the interval or the bound search, we link the upper bounds of the search with shortcut lower bounds as often as possible. The forward interval and bound searches may check whether the current lower bound minimum consumption exceeds a shortcuts recuperation gap. If so, we link only with the shortcut lower bound as during the preprocessing. The backward interval search cannot take advantage of recuperation gaps. During the backward bound search, whenever we link $s = (u, v)$ to the bounds at v we may at most use the information from the forward interval search. I.e. if u is a core vertex and was visited by the forward interval search, we are aware of the correct minimum consumption $\min_C$ required to reach u from s. If $\min_C$ exceeds the recuperation gap of s, then we may link with the lower bound of s.

In this context, a complication arises from the bound simplification. Consider a shortcut or an edge $s = (u, v)$ with a lower bound $\underline{s}$ and a simplified lower bound $\underline{\ell}$. Suppose we link the upper bound $\overline{f}$ at u with $\underline{\ell}$ during the relaxation of s. As $\underline{\ell}$ was simplified from $\underline{s}$, $\underline{\ell}$ is not an actual profile which we may realize in G. Furthermore, the simplification $\underline{\ell}$ is a lower bound of $\underline{s}$ and so the link $\overline{f} \circ \underline{\ell}$ is not a valid upper bound of $\overline{f} \circ \underline{s}$. To deal with this problem we store two simplifications of $\underline{s}$: a simplified lower bound $\underline{\ell}$ and simplified upper bound $\overline{\ell}$. We then link $\overline{f}$ with $\overline{\ell}$, obtaining a valid upper bound of $\overline{f} \circ \underline{s}$. In other words, we have two simplifications per shortcut lower bound, so that we may take advantage of recuperation gaps during the CH query.

6. Experimental Evaluation

In Chapters 4 and 5 we presented algorithms which compute consumption and travel time tradeoff profiles. Throughout this chapter we examine different aspects of these algorithms, and explain the results of our experimental evaluation. We then proceed with a case study of the tradeoff profiles we compute. Finally, we present details of our implementation.

6.1. Experiments

In this section we evaluate the algorithms introduced in Chapter 4 as well as their improvements discussed in Chapter 5. We start with an overview of the instances on which we run our measurements. We then measure the running times of the basic algorithms and discuss the results. Further we evaluate the quality of our CH and discuss different core sizes and simplified bound sizes. Lastly we offer comparison of the merge, link and domination operations performed by the standard profile algorithm.

Our implementations are written in C++ and compiled with the *GCC 4.8.3* compiler, optimization flags **-O3**. The running times are measured on a Dual 8-core Intel Xeon E5-2670 clocked at 2.6 GHz, with 64 GiB of DDR3-1600 RAM, 20 MiB of L3 and 256 KiB of L2 cache.

6.1.1. Problem Instances

We run our experiments on the street networks of Luxembourg and Germany, kindly provided to us by PTV AG[1]. We examine two instances of the Luxembourg network which is represented by a small graph. In our standard Luxembourg instance, denoted by *lux*, a third of the edges allow multiple tradeoffs and the rest of the edges may be traversed with only one pair of consumption and travel time. The conservative Luxembourg instance, which we denote by lux_c, only has 6 % tradeoff edges. The Germany instance (*ger*) has a more extensive graph than the Luxembourg ones, and is also conservative in terms of tradeoff edges. An overview of the three instances can be found in Table 6.1 on the following page. As Baum et al., we use altitude data provided by the CGIAR Consortium for Spatial Information[2].

As mentioned in Chapter 3, to generate the interpolation points of our tradeoff functions we use the consumption tables of Baum et al., [BDHS+14]. Specifically, each table entry of an

[1] http://www.ptvgroup.com/
[2] http://srtm.csi.cgiar.org/

Table 6.1.: Problem instances which we use throughout our experiments. The Luxembourg graph is small and nearly all s–t pairs result in reachable queries, given a 16 kWh battery. The Germany graph is much larger and most queries may not reach the target vertex with a 16 kWh battery. Like the second Luxembourg instance, lux_c, the Germany instance is also conservative w.r.t. edges which allow multiple tradeoffs.

Instance	Vertices	Edges	Tradeoff edges	Avg. complexity
lux	36 457	81 950	33 %	1.95
lux$_c$	36 457	81 950	6 %	1.20
ger	4 692 091	10 805 429	5 %	1.15

edge in their setting is an interpolation point of the edges cost function in our setting. The speed-consumption tables of Baum et al. are based on PHEM [HRZL09] measurements for the Peugeut iOn vehicle, without any auxiliary consumption (e.g. due to an air-conditioner). The standard Luxembourg and Germany instances we obtain without changing any tables. For the conservative Luxembourg instances we allow only a single driving speed for some road category types, sinking the average function size near that of the Germany instance.

For the travel time scale we choose seconds and for the energy consumption scale we choose mWh (milliwatt hours).

6.1.2. Algorithms Evaluation

We begin our evaluation by measuring the running times and results sizes of the algorithms discussed in Chapter 4, with the A* improvement of Chapter 5. The results we compare with those of the algorithm found in [BDHS$^+$14]. We continue the evaluation with rank plots of our basic algorithms, again with the A* goal direction. We then consider the running time and quality of our CH, both during the preprocessing and query phase.

For all our experiments we apply the A* improved target pruning to the basic profile algorithms. It is also important to note that we use the timestamps technique (Section 4.1) only during Luxembourg queries. In the case of Germany we clear the tradeoff function at any visited vertex after each query, which increases running time. If we do not, however, we run out of memory during the course of the queries: "outdated" functions at vertices consume too much memory and the machines 64 GiB become insufficient.

Random Queries. To measure running times of our basic algorithms we use a standard 16 kWh battery and run 1 000 *reachable* queries. The source and target (s and t) vertices we choose uniformly at random. Both our basic algorithms detect whether t is reachable from s during the initial potential computation of A*. In the case of an unreachable t, no actual profile search is performed and so query times consist solely of the potential computation. Unreachable queries are thus of little interest. For this reason we choose random vertex pairs until 1 000 queries succeed in reaching t, and take only measurements from these "successful" queries.

We were unable to run decisive random queries with a 60 kWh battery, which is also available at the present time. We encountered queries with prohibitive running time on the scale of hours. Later in this section we conduct rank queries with a fictive 1 MWh battery.

In the following the label-correcting standard profile algorithm is denoted by *Profile* and the label-setting algorithm is denoted by *MLProfile*. Aside from these two algorithms, we also run the multi-edge Pareto based approach of Baum et al., [BDHS$^+$14], using their

Table 6.2.: Average running times and result sizes of 1 000 random reachable queries on the Luxembourg and Germany instances, of our basic algorithms ("Profile", "MLProfile") with A* and the approach of Baum et al. ("Tradeoff", [BDHS+14]).

Algorithm	Solution size			Time [s]		
	lux	lux_c	ger	lux	lux_c	ger
Tradeoff	798	427	1 927	2.8	0.7	148.6
MLProfile	404	307	1 475	1.4	1.0	104.0
Profile	349	295	1 534	0.5	0.4	64.3

implementation. This algorithm we denote by *Tradeoff*. We run the same queries and battery capacity in order to compare running times and result sizes.

The average running times and result sizes of each algorithm are listed in Table 6.2. As seen, the standard profile approach produces compacter results (factor 1.3 to 2.3) in shorter times (factor 1.8 to 5.6) then the multi-edge Pareto approach of Baum et al. In the case of the two Luxembourg instances we also observe that the more edges with actual functions exist, the better our basic algorithms scale in comparison to [BDHS+14]: the Pareto algorithm takes 4 times longer in the presence of 5 times more tradeoff edges, whereas the running times of the MLProfile and Profile algorithms increase by a factor 1.4 resp. 1.3.

Note that in an instance where no edge allows a tradeoff, the profile algorithms function similarly to the algorithm of Baum et al. Thus the more conservative the instance is, the less beneficial the standard profile approach becomes. We also observe the difference of result sizes of the MLProfile and Profile algorithms, as discussed in Chapter 4.

Standard Profile Keys. As discussed in Chapter 4, the priority queue keys used during the standard profile algorithm have an impact on query times. To illustrate this, we run a 1 000 random reachable queries on the Luxembourg instance with a 16 kWh battery and five different keys. We measure running times, vertex and edge scans as well as the cost of the operation discussed in Chapter 3. During a domination test we count the number of iterations (lines **5** to **11**, Algorithm 3.5 on page 33). While applying the merge operation we count the iterations (lines **6** to **17**, Algorithm 3.1 on page 21) and also the intersections between the two functions (line **11**). During a half-convex link we count iterations (lines **6** to **16**, Algorithm 3.3 on page 28) and secondary iterations (lines **10** to **16**). As before, we increment the counted values by 1 due to unreachable queries or cases where an operation was not performed. The results we compare to the label setting algorithm, MLProfile.

For the standard algorithm we apply the observations from Section 4.1, i.e. after a merge we use the travel time of the last intersection point the key denoted by *key point*. Specifically, we use lexicographical sorting where travel time takes precedence. Suppose vertices u and v with functions f_u and f_v, having key points $p_u = (\tau_u, \mathrm{C}_u)$ resp. $p_v = (\tau_v, \mathrm{C}_v)$, and reside in the priority queue. The key of u is strictly less than that of u, iff $\tau_u < \tau_v$ or $\tau_u = \tau_v \wedge \mathrm{C}_v < \mathrm{C}_v$.

The second key type also uses the key point of functions, however it uses a linear combination of the travel time and consumption values: $0.5 \cdot \tau + 0.5 \cdot \mathrm{C}$. The third and fourth type use the last resp. first point of functions as the key point with lexicographical order. For the last key type, we examine the minimum travel time and consumption as key, also with lexicographical order.

In the same order of key usage, the average running times of the standard profile algorithm can be seen in Table 6.3 on the next page. As discussed in Section 4.1, choosing an appropriate priority queue key leads to performance better than that of the label setting

Table 6.3.: Comparison of the standard profile algorithm with different priority queue keys and the label setting algorithm. The averages from 1 000 random reachable queries on the standard Luxembourg instance with a 16 kWh battery are listed.

Key type	Sorting	Time [ms]	Vertex scans	Edge scans	Merge cost	Link cost	Domin. cost
MLProfile	—	1 426	85 648	180 437	$1.2 \cdot 10^7$	$1.2 \cdot 10^6$	$6.2 \cdot 10^6$
key point	lexicogr.	532	14 998	32 489	$2.8 \cdot 10^6$	$5.6 \cdot 10^6$	$9.1 \cdot 10^6$
key point	lin. comb.	1 079	20 826	45 833	$6.5 \cdot 10^6$	$1.0 \cdot 10^7$	$1.6 \cdot 10^7$
$(\max_\tau, \min_C)$	lexicogr.	891	20 371	45 553	$4.1 \cdot 10^6$	$8.8 \cdot 10^6$	$1.4 \cdot 10^7$
$(\min_\tau, \max_C)$	lexicogr.	3 867	42 825	96 370	$2.1 \cdot 10^7$	$3.6 \cdot 10^7$	$7.7 \cdot 10^7$
$(\min_\tau, \min_C)$	lexicogr.	2 465	41 500	93 275	$2.0 \cdot 10^7$	$3.5 \cdot 10^7$	$7.4 \cdot 10^7$

algorithm. Choosing the key point as basis for the keys yields the best performance for the standard profile algorithm. On the other hand, a key which poorly reflects the change in functions after a merge, such as the minimum point of the merge result, slows down the algorithm. This is due to the improved target pruning with A*, Section 5.1. This we observe in the number of vertex and edge scans: the sooner a "good" profile is propagated to t, the more functions at vertices can be discarded with the improved target pruning. The better we choose our priority queue keys, the earlier such a profile is present at t.

Ranks. For more detailed information on the running times of our algorithms, we run rank queries. In the standard setting of Dijkstra's algorithm, the vertex ranks are equal to the vertex setting order. I.e. the first settled vertex, s, has rank 0, the second settled vertex has rank 1 and so on. We choose the same vertex ranks with travel time as the scalar criterion for Dijkstra. We run a 100 Dijkstra searches with a source vertex chosen uniformly at random and travel time as criterion, until all vertices are settled. For each run we take the vertices with rank $2^5, 2^6, 2^7$ and so on. The 100 one-to-one profile queries for a specific rank i then consist of the 100 random source vertices, and their respective vertex with rank 2^i as target. As with the random queries, we run the rank queries with a 16 kWh battery.

Figure 6.1 on the facing page shows the rank plots for the standard Luxembourg and the Germany instances with a 16 kWh battery. In both plots we observe the exponential growth of running time as the query rank increases. Until ranks 2^{12} for Luxembourg and 2^{15} for Germany the running time of the algorithms is dominated by the scalar searches done by A*, which compute the vertex potentials. In the Luxembourg instance no drop in running time can be seen, as 16 kWh are sufficient to reach nearly any vertex from a given source s. In the Germany instance this is no longer the case and after rank 2^{18} the running times begin to improve. From rank 2^{21} on the running times consist solely of the A* scalar searches: as soon as the minimum consumption backward search from t is unable to reach s, we know that no s–t path is traversable and the query stops.

CH Preprocessing. We continue the evaluation with the preprocessing phase of our CH. Here we use running times on the standard Luxembourg instance and the Germany one. In the following, the core size denotes the percentage of uncontracted vertices. During the preprocessing phase, we choose to use potentials during our profile witness searches (Sections 5.2.1 and 5.2.1) after the core size reaches 10 %, for both instances. We parallelize our witness searches (Section 5.2.1) after 10 % core in the Luxembourg instance, and after 33 % in the Germany one.

Table 6.4 on page 62 shows the statistics of the preprocessing phase. At different core sizes the table lists the averages of the following metrics. The second and third columns contain

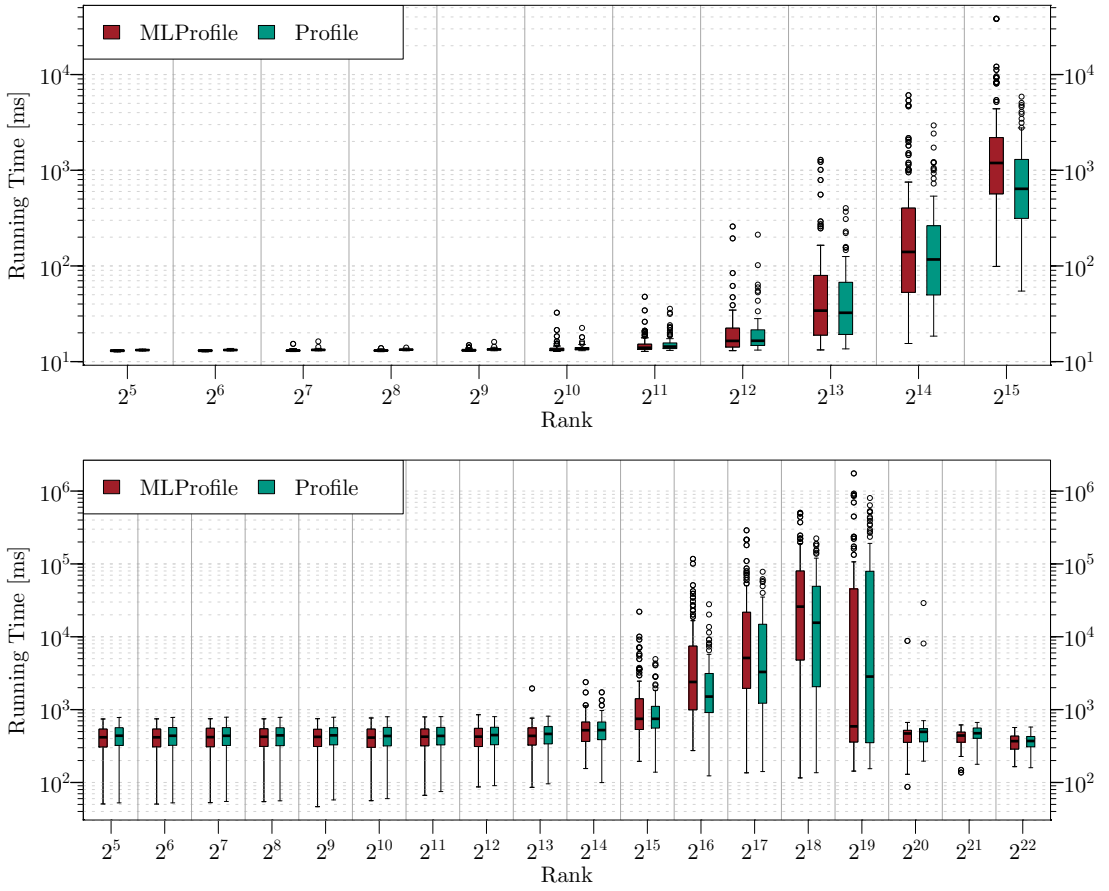


Figure 6.1.: Rank queries of our basic algorithms with A*, 16 kWh battery. The results for the standard Luxembourg (lux) instance can be seen above, and those for the Germany instance (ger) – below.

the function complexity and the maximum recuperation gap (in Wh) of core edges and shortcuts. The fourth column lists the core degree. The sixth and seventh columns give the number of unpacked edges and the function complexity of *all* edges and shortcuts in the contracted graph G'. The eight and ninth columns represent the same values, but only for *shortcuts*. The last columns do not hold average values, but contain the ratio shortcuts (and edges) in G' to original edges of G, and the running time of the CH preprocessing so far.

As seen, for all core sizes the overall function complexity and number of unpacked edges is small. The recuperation gap (Chapter 5) of the Luxembourg core is higher than that of the Germany one, as the original edges of the Luxembourg instance allow more recuperation: 2 Wh on average, compared to 0.1 Wh in the case of Germany. The total number of shortcuts and edges in the contracted graph also remains below twice the number of original edges in G. Furthermore, all of the mentioned values so far do not increase highly between core size milestones. This is not the case for the core function complexity, the hierarchy depth and the core degree. In the Luxembourg instance the complexity doubles from 2 % to 1 % and triples from 2 % to 0.5 %, however we are still able to contract the graph fully. In the case of Germany, the complexity doubles and then increases five-fold during the transitions from 2 % to 1 % and from 2 % to 0.5 %. Combined with the steep increase of core degree, contracting the remaining core vertices becomes very expensive – as indicated by the time required to contract the Germany graph from 2 % to 0.5 %. Fully contracting the Germany instance thus becomes infeasible.

Table 6.4.: Measurements during the CH preprocessing phase at specific core sizes. The first four columns offer statistics of the core graph. The fifth to tenth columns relate to the whole contracted graph. The last column indicates the total running time of the preprocessing. Columns 2 to 4 list averages of function complexity, recuperation gap and degree in core. Columns 6 and 7 contain average overall complexity and number of unpacked edges of the contracted graph. Columns 8 and 9 show the same values, but only for shortcuts. Column 10 lists the ratio of shortcuts and edges in the contracted graph to original edges.

	Core size	Core cplx.	Core gap	Core deg	CH depth	Avg. unpk.	Avg. cplx.	Shct. unpk.	Shct. cplx.	Shct. quot.	Time
lux	2.00 %	23.6	9.53	6.68	17	3.57	5.82	6.80	9.26	1.80	9 s
	1.00 %	42.3	9.58	9.22	23	4.73	7.12	9.13	11.85	1.84	14 s
	0.50 %	64.2	9.82	12.45	29	6.18	8.48	12.14	14.60	1.87	30 s
	0.00 %	—	—	—	79	8.71	10.66	17.38	19.10	1.89	70 s
ger	2.00 %	11.6	2.71	6.80	18	3.24	3.13	6.15	4.38	1.77	22 m
	1.00 %	26.0	2.97	9.75	23	4.35	3.93	8.49	6.11	1.81	46 m
	0.50 %	62.8	3.44	14.70	31	6.29	5.62	12.58	9.73	1.84	223 m
	0.33 %	111.2	3.70	19.11	37	8.42	7.57	16.68	13.68	1.86	1 031 m

In terms of size our CH preprocessing consumes 5 to 7 times more storage than the original graphs, for both the standard Luxembourg and Germany instances. Here we examine the resulting memory consumption after simplifying the bounds of edges and shortcuts (Chapter 5). A fully contracted, simplified to size 8 Luxembourg instance requires 34.2 MiB, up from 5.2 MiB. The Germany instance contracted to 0.5 %, again with simplification size 4, grows to 3.3 GiB from 0.6 GiB.

CH Query. We proceed with the query phase, by running the exact same rank queries as the ones we use to plot Figure 6.1 on the preceding page. As core size we use 0 % in the case of Luxembourg and 0.5 % for Germany. We simplify bounds so that tradeoff functions have at most 8 and 4 interpolation points, for Luxembourg resp. Germany.

The plots seen in Figure 6.2 on the next page illustrate the speed-up yielded by our CH adaptation (denoted by *CHProfile*), against the standard profile with A*. It is evident that the CH does not offer a significant speed-up. Two oracles can be seen in the same plot, which we use to explain the low speed-up. The one denoted by *PerfectOracle* is the standard profile algorithm, running on the solution space – only the edges lying on paths which construct the tradeoff profile at t. The *A*Oracle* functions in the same manner, but uses all edges which were considered by the standard profile algorithm during the s–t query.

The A* oracle rank queries yield the running time of the standard profile, without the potential computation. This allows us to compare speed-ups at low ranks, where the potential computation dominates running time. The perfect oracle offers the maximum speed-up which we may achieve with edge pruning: it only considers edges which are contained in the result. While the CH adaptation does not speed-up the queries significantly, this is also not achieved by the maximum speed-up algorithm. In other words, the profile results are a bottleneck for any edge pruning algorithm – the s–t profiles contain too many paths.

Search Spaces. To better understand the poor speed-ups via edge pruning we also plot the search space sizes of the perfect oracle, A* oracle and the CHProfile algorithm. Here, we divide the CHProfile into two portions: space sizes offered by the interval and bound searches. We use the same queries as in the previous paragraph, however we do not simplify

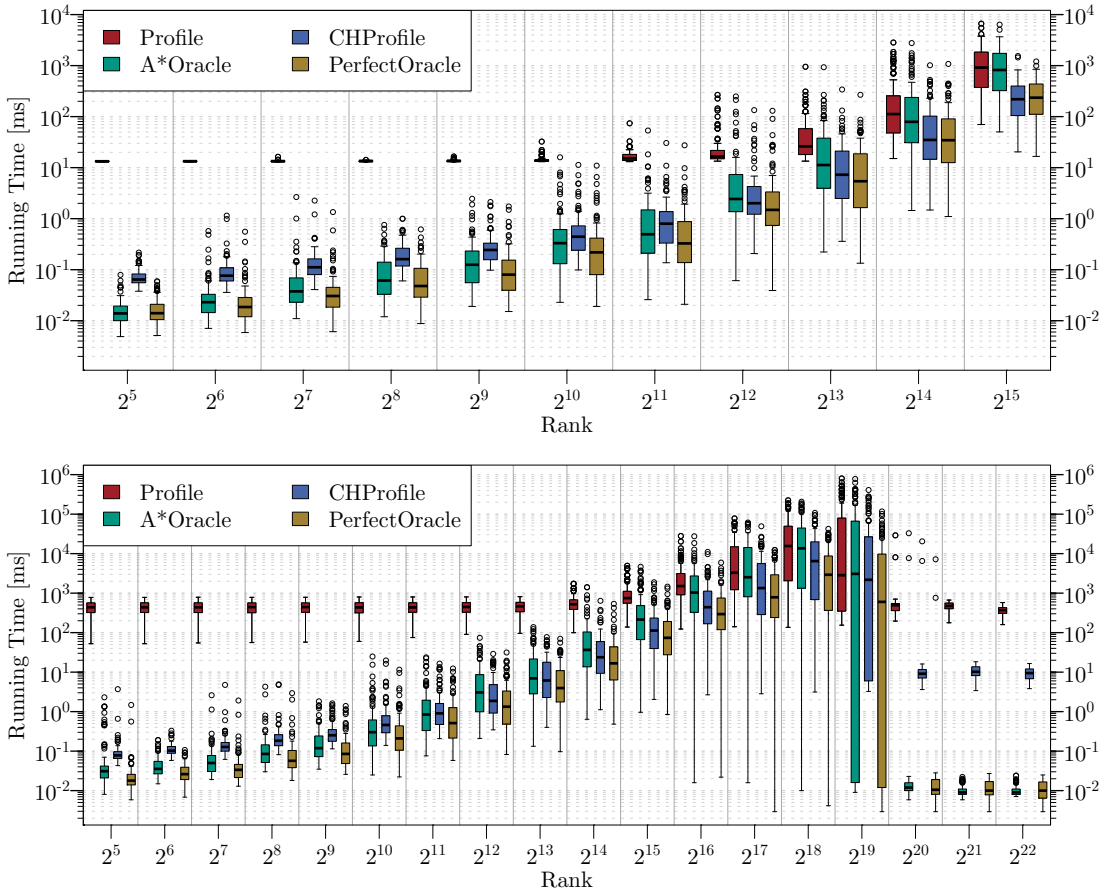


Figure 6.2.: Running times of of our CH adaptation. As before the Luxembourg plot is seen above, and the Germany one below. The CHProfile is compared to the standard profile and two fictive algorithms. The A* oracle considers only edges relaxed by the standard profile algorithm. The perfect oracle ignores all edges which do not lie on paths improving the s–t profile.

bounds at any time. We do so to illustrate the quality of the contraction hierarchy. See the following paragraph for the effect of simplified bound and core sizes on the resulting search spaces, as well as on the CH query running times.

Note that in the case of unreachable t from s, the pruning algorithms yield empty search spaces. As the y-axis of our plots is logarithmic, we increment all search space sizes of the Germany queries by 1 before plotting.

Figure 6.3 on the following page shows the resulting plots. As stated in Section 5.2, due to the shape of our tradeoff functions the interval search is not able to prune well. This is amended by the following bound search, which uses the complete CH bounds. On average, the bound search yields no more than two times larger search spaces than those of the perfect oracle. We also observe that the search space of the A* technique is only an order of magnitude larger than the optimal search space.

Core and Simplified Bounds Sizes. While the search spaces produced by a CH query without bound simplification are small, the bound search is too costly (Table 6.5). We thus explore different simplification sizes and also different core sizes, in order to find a sweet spot for our CH queries.

To identify a core and simplification size sweet spot we examine 1 000 random reachable queries on the Luxembourg instance, as there we are able to contract fully during the

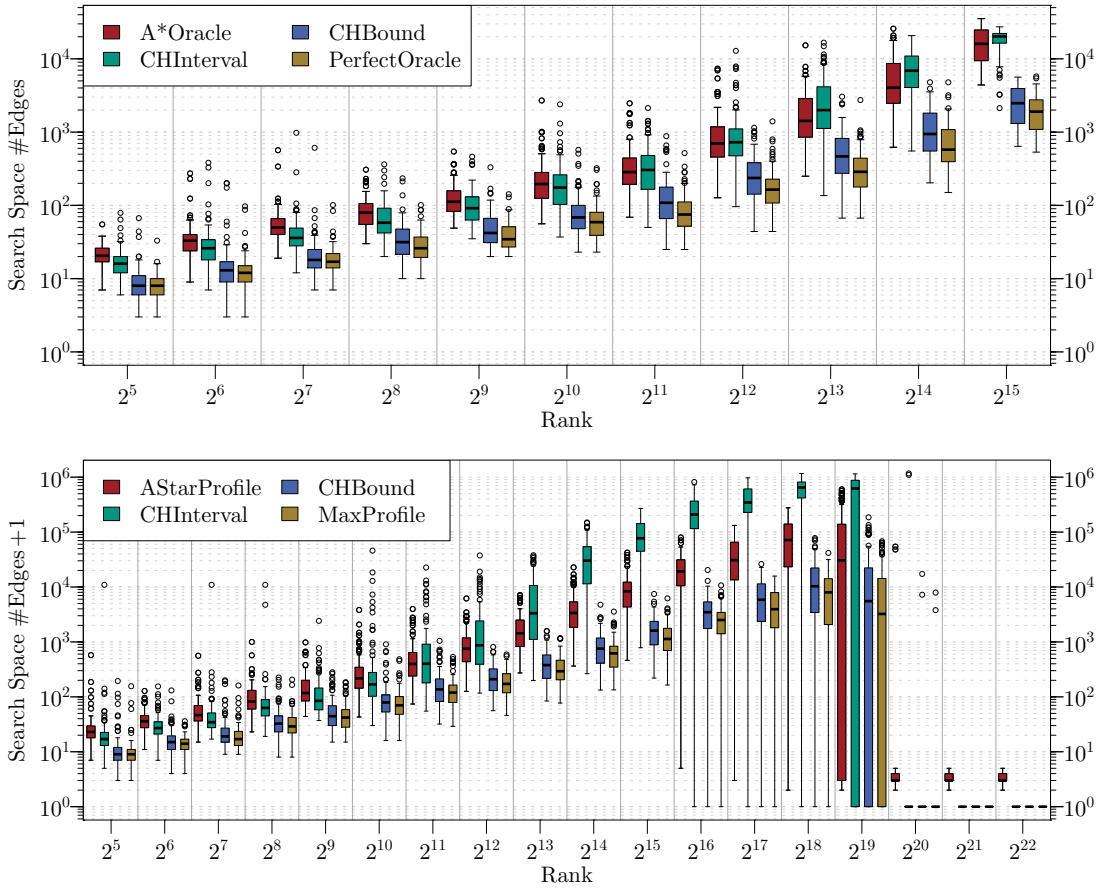


Figure 6.3.: Edge search spaces of the "speed-up" algorithms compared in Figure 6.2. The CHProfile consists of an interval search and a bound search, both of which are included in the plot. The CHProfile search space equals that of the bound search.

preprocessing. Here we measure the CH query time and the number of edges in the search space yielded by the bound search. Table 6.5 on the next page shows the average values for different combinations of core sizes and simplification sizes. As seen the CH query benefits most from the fully contracted graph, since the running times and search space sizes degrade with the increase of the core size. We also observe that a simplification size of 8 is optimal in all cases but the 2 % core, where the bound search is slowed by the larger core and function sizes. Furthermore he differences in running time and search space size are not substantial, as long as we are applying simplification. The search spaces are smallest when we do not simplify at all (denoted by ∞), however the query times more than double due to the expensive bound search.

Next we run 100 random reachable queries on the Germany instance, but only with a 0.5 % core size. Instead of measuring the total CH query running times, we examine just phases of the CH query: the interval, bound and profiles searches, as well as unpacking the necessary shortcuts. As with the Luxembourg instance we also measure the search spaces yielded by the bound search. Table 6.6 on page 66 contains the measured times and sizes. Clearly, without simplification a CH query on the Germany instance is not feasible. We also note that the optimal simplification size is 4, and not 8 as with the standard Luxembourg instance, due to the large core. As in the case of Luxembourg the number of unpacked edges does not differ greatly when we use simplification, however a bound search in the core graph becomes more expensive the less we simplify.

Table 6.5.: A 1 000 random reachable queries on the standard Luxembourg instance, contracted to different core sizes. Each line represents a specific simplification size during the CH bound search, where ∞ denotes no simplification. The average CH query times and the edge search spaces yielded by the bound search are listed for each combination of core and simplification size.

	Core 0 %		Core 0.5 %		Core 1 %		Core 2 %	
Smpl. size	Time [ms]	Search Space	Time [ms]	Search Space	Time [ms]	Search Space	Time [ms]	Search Space
4	154.8	1 742	217.9	2 220	260.5	2 639	334.7	3 462
8	149.4	1 569	214.0	2 082	260.1	2 516	336.2	3 347
16	150.3	1 548	221.3	2 077	269.8	2 515	345.9	3 341
∞	381.5	1 311	625.8	1 580	841.6	1 746	1169.0	2 212

No Undercharging. We also run rank queries with a fictive 1 MWh (megawatt hour) battery on the Germany instance, allowing any vertex to be reached. As with the 60 kWh battery during our random queries, we encounter running times in the scale of hours. Whenever a query takes more than an hour, we invalidate the current rank measurements and do not run queries for higher ranks. The running times we measure in this manner are upper bounds, as we ignore any pruning due to undercharging. I.e. in this setting we give the worst running times of our algorithms, for each rank where no query took longer than an hour.

Note that running times of more than an hour may be introduced by memory swapping, as the 64 GiB of memory may be insufficient to contain the tradeoff functions to all vertices in the graph.

Figure 6.4a on page 67 shows the result of the queries. As seen we are only able run the first ranks 2^5 to 2^{19} with a 1 MWh battery, before a query takes more than an hour. We also no longer observe a drop in running times after rank 2^{18}, as opposed to Figure 6.2. The speed-ups we measure here are otherwise similar to those with a 16 kWh battery on the Germany instance, Figure 6.2 on page 63.

Memory Consumption. As noted previously, when running queries with a 1 MWh battery it is possible that we encounter memory swapping. I.e. during a query with the standard profile algorithm, the available memory is insufficient to store the profile functions at the vertices. We therefore also measure the summed sizes of these functions after a rank query finishes.

The results are depicted in Figure 6.4b on page 67. Similar to the running time, the summed function sizes at all vertices after a query increase exponentially with query rank. Already at rank 2^{19} we observe runaways where the query stores a total of over 10^9 interpolation points. As the progression seen in the plot implies, at rank 2^{20} a runaway no longer fits in the 64 GiB memory. As swapping is initiated, we stop the experiment. Note that even without any auxiliary data structures, storing a total of 10^{10} interpolation points already requires 70 GiB of memory – if a point is represented by two integers of 4 bytes each.

Operation Costs. Before concluding our experimental evaluation, we also examine the costs of the operations during a profile query as defined in Chapter 3. We run rank queries (16 kWh battery) with the standard profile algorithm and count the operations performed during linking, merging and domination tests. We count in the same manner as during our evaluation of the different priority queue keys for the standard profile algorithm.

The resulting plots can be seen in Figure 6.5 on page 68. The number of performed operations is asymptotically similar, where the domination routine requires the most

Table 6.6.: A 100 random reachable queries on the Germany instance, contracted to a 0.5 % core. Simplifying during the bound search to smaller function sizes yields bound searches which cost less, and do not prune significantly fewer edges.

Simplified size	Interval search [s]	Bound search [s]	Profile search [s]	Unpacking time [s]	Search space #edges
4	0.011	2.7	26.5	0.008	42 302
8	0.012	4.0	26.6	0.008	42 615
16	0.013	5.5	26.2	0.008	42 079
32	0.023	7.3	27.0	0.010	43 464
∞	0.015	493.1	12.7	0.002	17 693

operations and the merge one – the least. For future work it would therefore be most beneficial to improve first domination tests, then linking and finally merging, in case we choose to invest time in optimizing routines. Furthermore, although the half-convex linking is quadratic in the worst case, we do not observe any spike in the number of linking operations.

No Recuperation. Finally, we measure the running times of standard CH queries. So far we have sped our queries up by applying edge pruning, i.e. our CH query prunes provably unimportant edges for the search. During the actual query we do not take advantage of shortcuts, as done in a standard time-dependent CH query. And as seen, an edge pruning approach faces a bottleneck due to the result size. We therefore wish to explore a case where we may utilize shortcut functions.

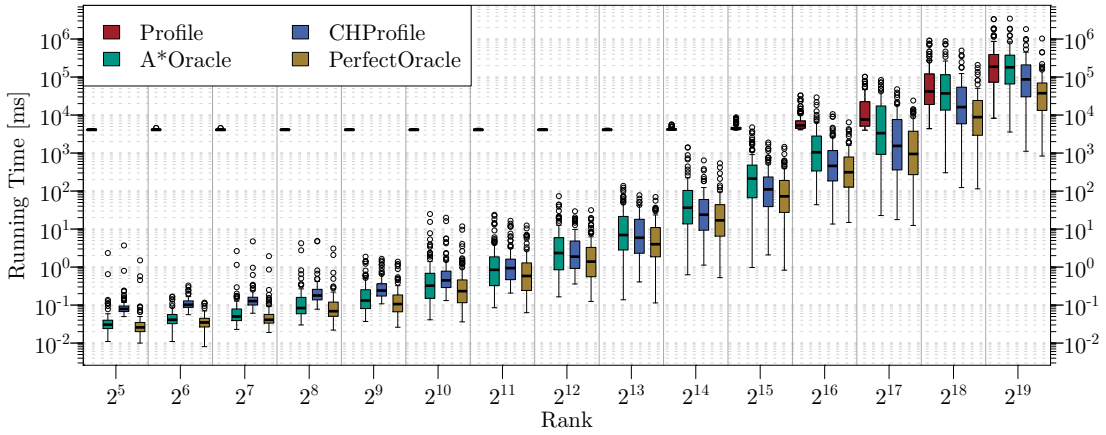
We do so by forbidding recuperation on the input instance. In this manner our preprocessing approach computes equal upper and lower bounds for each shortcut. In other words, the shortcut bounds represent the actual shortcut function and we may use these to run a standard time-dependent CH query, see [BGNS10] Algorithm 5.

The results can be found in Table 6.7 on page 69. As seen, forbidding recuperation to allow standard CH queries also does not drastically lower running times in the case of the Luxembourg instance. To explain this we examine the operations performed during the queries, in the same manner as in the last paragraph. We observe that very few vertices are scanned and few edges are relaxed. However the performed merge and link operations are nearly as costly, and so the speed-up is not high. In the case of the German instance, we observe no improvement in the query running times. As seen, the operation costs of the standard CH query are higher than those of the standard profile algorithm. The reason for this is the large core of the CH and the lack of A* goal direction. In other words, the uncontracted core is too costly.

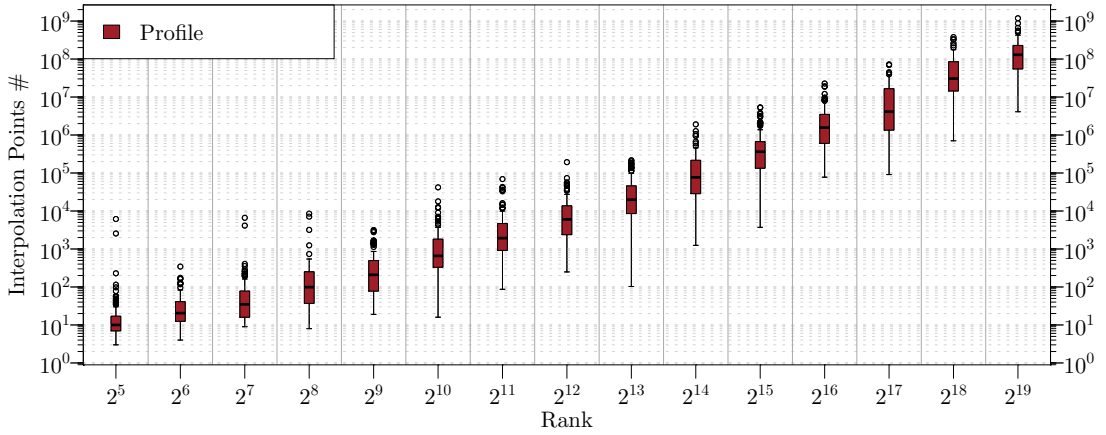
6.2. Case Study

We conclude the evaluation chapter with an example of the consumption and travel time profiles. For this we choose the standard Luxembourg instance, which allows more tradeoff edges. As during the experiments we use a standard 16 kWh battery. The profile we wish to present may be seen in Figure 6.6 on page 69.

As usual, we have both a fastest route and an energy-optimal route. We may reach the destination after 65 minutes by consuming 11 kWh, or we may invest 87 minutes and spend only 7 kWh. Our options however are not limited to these two routes. We identify a second route which is 2 minutes slower than the fastest route but requires 8.5 kWh, see Figure



(a) Running time comparison of our CH query, as in Figure 6.2.



(b) Total interpolation points of cost functions at all vertices, after query termination.

Figure 6.4.: Two experiments with a fictive 1 MWh battery on the Germany instance.

6.6c. A further route is 11 minutes faster than the energy-optimal route and costs only 0.2 kWh more.

We also see the tradeoff profile in Figure 6.6, where the grey line indicates which tradeoff point is achieved by traversing the respective route. The form of the profile is of particular interest, and we distinguish between three types of forms. First, the most beneficial profile shape is a *convex* curve with maximal length. In such a case we may resort to routes which are both fast and energy efficient. Second, if the profile shape is that of a diagonal *line*, no tradeoff sweet spots exist but we may still meaningfully invest travel time to lower consumption. And third, a shape of a *concave* curve with maximal length would imply that no sensible tradeoffs exist: investing more travel time would not lower consumption significantly. The profile shapes we observed were only variations of the first two types; none were of the last type.

6.3. Implementation Details

Next, we present our choice in interpolation point representations. Although of little consequence from theoretical standpoint, this choice plays an important role when implementing the purposed algorithms. For severe numerical issues and our solutions see the Appendix, Section A.

Tradeoff Function Representation. We choose to represent the interpolation points of the tradeoff functions with integer values. This decision is based on experience in time-

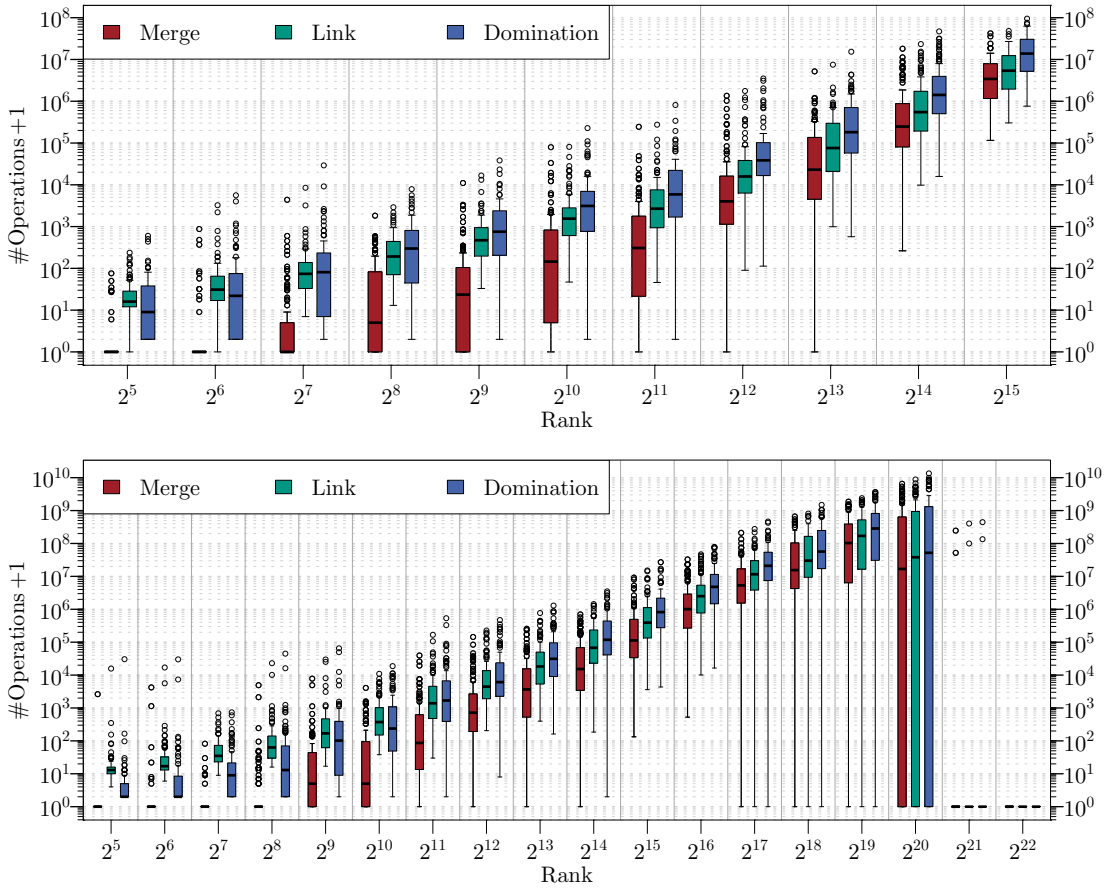


Figure 6.5.: Asymptotic costs of the operations performed by the standard profile algorithm. For each iteration of the routines in Chapter 3 a cost of one is counted. During the merge and non-convex linking routines, an additional cost of one is added for each computed intersection.

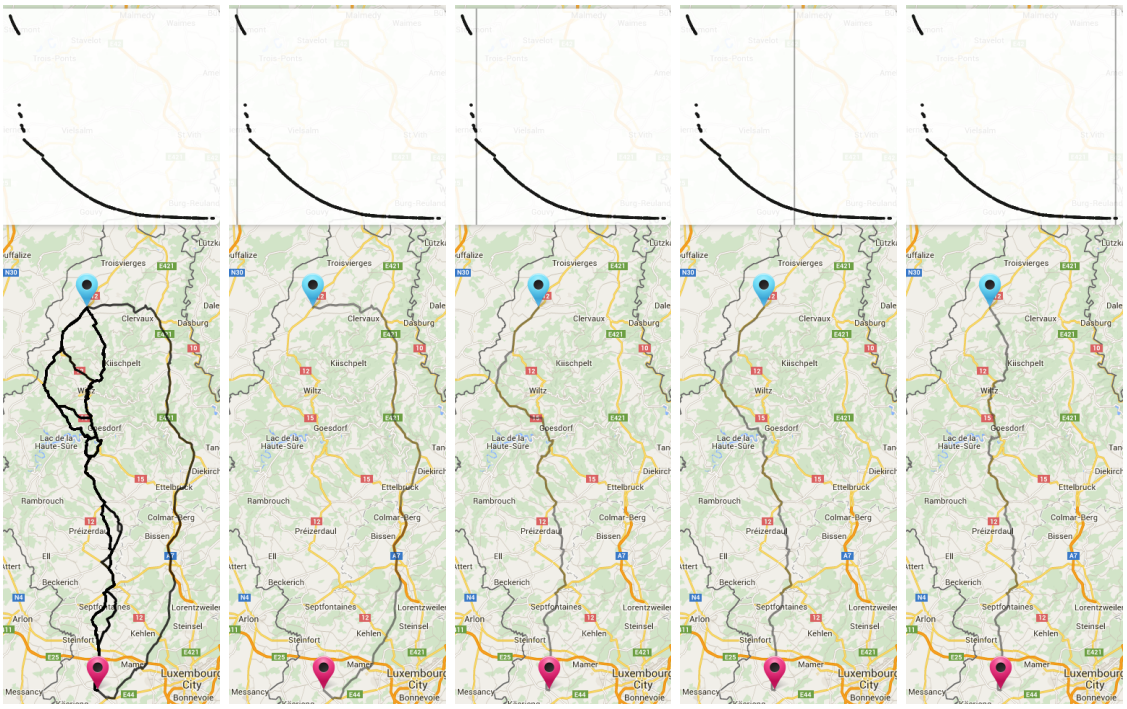
dependent routing, as double precision representation involves a number of problems such as numerical singularities. We attempt to avoid these issues with an integer representation.

The first consequence of this choice is the rounding of segment intersections. Obviously, tradeoff functions may intersect at non-integer values. Rounding such intersections down, so that we may represent them, would introduce values which we may not construct given the input functions. So instead we round both travel time and energy consumption up. Of course, we still introduce an error to the result. Within a single merge, the merged function deviates from the actual result by at most (euclidean) distance 1. Depending on slope, however, the actual travel time and consumption differences may be substantial. Whenever the slope is steep the travel time difference is small and the consumption difference large. The opposite occurs when the slope is nearly 0.

The limited error of a single merge does not imply that the result error is also limited. Successive n merges may introduce a total distance of n to the actual result. However the average overall error remains small, due to the vast span of the resulting functions. Table 6.8 on page 70 shows average error distances of the standard profile and label setting algorithms. We first use the result $\mathcal{B}(t)$ computed by the tradeoff algorithm of Baum et al. as a reference. For each non-dominated (by our computed profile) tradeoff point $p \in \mathcal{B}(t)$ we compute the euclidean distance from p to f_t, which results from either the standard profile or the label setting algorithm. We then sum the total such distance and divide it by

Table 6.7.: Standard CH queries compared to the standard profile algorithm. During the CH queries recuperation is forbidden, which diminishes the computed profiles.

	Algorithm	Core size	Time [s]	Vertex scans	Edge rlxs.	Merge cost	Link cost	Domin. cost
lux	Profile	—	0.54	14 998	17 573	$2.8 \cdot 10^6$	$5.6 \cdot 10^6$	$9.1 \cdot 10^6$
	CH	0.5 %	0.24	275	2035	$2.3 \cdot 10^6$	$3.6 \cdot 10^6$	$3.8 \cdot 10^5$
	CH	0.0 %	0.16	196	1263	$1.6 \cdot 10^6$	$2.2 \cdot 10^6$	$9.7 \cdot 10^4$
ger	Profile	—	64.3	50 334	579 563	$5.4 \cdot 10^9$	$7.2 \cdot 10^9$	$1.1 \cdot 10^{10}$
	CH	0.5 %	97.8	6 792	85 423	$9.9 \cdot 10^9$	$1.8 \cdot 10^{10}$	$1.6 \cdot 10^9$



(a) (b) (c) (d) (e)

Figure 6.6.: A profile query on the standard Luxembourg instance with a 16 kWh battery. All routes from the origin to destination may be seen in (a). The fastest route (b) requires 65 minutes and 69 % of the battery. A marginally slower route (c) takes 2 more minutes but only 53 % battery. An energy efficient route (d) requires 76 minutes and 44 % battery. The energy-optimal route (e) in contrast costs 87 minutes and consumes 43 % of the battery.

the number of points in $\mathcal{B}(t)$, obtaining the average error distance per tradeoff point. As seen, the error introduced in the result is not large.

Second, due to our choice of function representation, we may test for domination without precision loss. Algorithm 3.5, which checks if $f \propto g$, must test whether a point (τ, C) lies above a segment s. For this we interpolate the consumption of s at τ by using the slope of s, and compare the interpolated value with C. As consecutive interpolation points are represented by integer values, the slopes of tradeoff functions are rational numbers. We may therefore transform the inequality into a division free form and work with full precision.

Third, during the CH preprocessing rounding is less of an issue as we work with bounds. We note that we round the intersections of upper and lower bounds up resp. down. The

Table 6.8.: Total average error (euclidean) distance d of 100 queries with 16 kWh of the basic profile algorithms. The first column shows the error of the basic algorithms with the tradeoff algorithm of Baum et al. as a reference. The second and third columns show the error when comparing the two basic algorithms with one other.

	Algorithm	Tradeoff	Profile	MLProfile
lux	Profile	0.007	—	0.005
	MLProfile	0.010	0.006	—
ger	Profile	0.007	—	0.004
	MLProfile	0.007	0.003	—

actual profile query is run on a sub-graph of the original graph and so the only error is the one introduced by the basic profile algorithm.

7. Conclusion

In this thesis we examined energy consumption and travel time profiles in electric vehicle routing. These profiles represent the tradeoff between consumption and travel time, which is essential when searching for fast and energy efficient routes from origin to destination.

We first devised edge cost functions which reflect the consumption and travel time tradeoff of traversing street segments with different speeds. As a representation for these tradeoff functions we chose piecewise linear functions. To enable profile queries with Dijkstra's algorithm we adopted the merge operation and the domination test found in time-dependent routing. We then defined the link operation and devised an algorithm which computes the link result of two tradeoff functions. Here, we distinguished three cases of increasing computational complexity, where we take advantage of convexity in the simpler cases.

Furthermore, we adopted the time-dependent profile Dijkstra as a label-correcting approach. We also offered a label-setting algorithm, which propagates convex sub-functions instead of whole functions. For these basic algorithms we adapted the A* technique in order to handle negative consumption costs, to allow target pruning and to direct the search. In addition, we devised a method of deriving user speed directions in the case of the label-setting algorithm.

To speed up the profile searches further we examined the time-dependent adaptation of the CH algorithm. We refitted the preprocessing phase of this CH variant, to remedy the problem of state-of-charge dependent shortcut profiles. Specifically, we utilized shortcut bounds and adjusted the preprocessing to work with these. We introduced a new term to the vertex contraction costs, which keeps the bounds tight. We also took advantage of path prefixes, to further minimize the distance between lower and upper bounds. For the query phase we devised a new interval search and introduced on-the-fly simplification during the bound search.

Finally, we evaluated our algorithms experimentally. We compared our basic algorithms with an existing approach in terms of running time and solution size. We then examined the speed-ups of our CH adaptation and identified a bottleneck of edge pruning based speed-up techniques in our scenario. For this we compared search space sizes of our algorithms and an optimal oracle. Furthermore we examined the memory consumption and performed operations of the standard profile algorithm. And last we discussed the representation we chose for the interpolation points of our profile functions.

Overall, the profiles we discussed offer important information such as the existence or lack of fast and energy efficient routes. Compared to previous works which consider the tradeoff

between energy consumption and travel time, our profile approach yields exact queries which run faster and allow a wider range of driving speeds per street segment. And last, in our setting we observed that the CH speed-up technique is not far superior to the A* goal direction.

Future Work

Possible future work of this thesis branches out in several directions. First, consumption and travel time profiles are not restricted to piecewise linear functions. In [Zün14] hyperbolas are briefly examined for the modelling of charging functions. Zündorf observes that linking hyperbolic functions may be done in constant time, a fact from which our label-setting algorithm may benefit. Exploring other function representations may likely be beneficial.

Second, the speed-ups we achieve in this thesis are not sufficient. To improve query times a technique which uses shortcut profiles to skip vertices, and not only to prune edges, is likely required. A possible approach would be to utilize shortcut profiles where both travel time and the state of charge are a parameter. If it is possible to avoid complex merge and link operations which work in $\mathbb{R}^2$, such an approach may be feasible.

Moreover, charging stations may be integrated to allow greater cruising ranges. Naively, a loop may be introduced with a function which allows "charging" by mapping charging time to negative consumption. As seen in our evaluation, extensive cruising range quickly leads to large memory consumption and very long query times. Thus if charging is introduced to the algorithms, a suitable speed-up technique is required.

The modelling may also be refined by taking speed transition costs into account, for instance turn costs integration. Our linear interpolation involves traversing a street segment with two different speeds. The transition may therefore cost extra energy, or allow energy recuperation to some extent. Taking this into account would lead to more accurate and complete tradeoff profiles.

And last, heuristics may be explored. As observed, query times are not suitable for an interactive application and in no way sufficiently low for a server scenario. A likely viable heuristic is to use function simplification, in order to retain the general form of the tradeoff but greatly reduce the number of used interpolation points. Another heuristic of interest may compute only the fastest feasible route from origin to destination, while still taking variable driving speed into account. E.g. the label setting approach may stop the query as soon as the target vertex is reached. Building upon this idea may then yield feasible query times, possibly even in the server scenario.

Acknowledgment

We would like to thank Strasser for identifying the quadratic worst case example used in the proof of Lemma 3.9.

Bibliography

[BDHS+14] Moritz Baum, Julian Dibbelt, Lorenz Hübschle-Schneider, Thomas Pajor, and Dorothea Wagner. Speed-consumption tradeoff for electric vehicle route planning. In Stefan Funke and Matúš Mihalák, editors, *14th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems*, volume 42 of *OpenAccess Series in Informatics (OASIcs)*, pages 138–151, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[BDPW13] Moritz Baum, Julian Dibbelt, Thomas Pajor, and Dorothea Wagner. Energy-optimal routes for electric vehicles. In *Proceedings of the 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL'13, pages 54–63, New York, NY, USA, 2013. ACM.

[BDS+08] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining hierarchical and goal-directed speed-up techniques for dijkstra's algorithm. In CatherineC. McGeoch, editor, *Experimental Algorithms*, volume 5038 of *Lecture Notes in Computer Science*, pages 303–318. Springer Berlin Heidelberg, 2008.

[Bel56] Richard Bellman. On a routing problem. Technical report, DTIC Document, 1956.

[BGNS10] Gernot Veit Batz, Robert Geisberger, Sabine Neubauer, and Peter Sanders. Time-dependent contraction hierarchies and approximation. In Paola Festa, editor, *Experimental Algorithms*, volume 6049 of *Lecture Notes in Computer Science*, pages 166–177. Springer Berlin Heidelberg, 2010.

[CM85] H.W. Corley and I.D. Moon. Shortest paths in networks with vector weights. *Journal of Optimization Theory and Applications*, 46(1):79–86, 1985.

[DGNW13] Daniel Delling, Andrew V Goldberg, Andreas Nowatzyk, and Renato F Werneck. Phast: Hardware-accelerated shortest path trees. *Journal of Parallel and Distributed Computing*, 73(7):940–952, 2013.

[DGPW11] Daniel Delling, AndrewV. Goldberg, Thomas Pajor, and RenatoF. Werneck. Customizable route planning. In PanosM. Pardalos and Steffen Rebennack, editors, *Experimental Algorithms*, volume 6630 of *Lecture Notes in Computer Science*, pages 376–387. Springer Berlin Heidelberg, 2011.

[Dij59] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[DSW14] Julian Dibbelt, Ben Strasser, and Dorothea Wagner. Customizable contraction hierarchies. In Joachim Gudmundsson and Jyrki Katajainen, editors, *Experimental Algorithms*, volume 8504 of *Lecture Notes in Computer Science*, pages 271–282. Springer International Publishing, 2014.

[DW09] Daniel Delling and Dorothea Wagner. Time-Dependent Route Planning. In RavindraK. Ahuja, RolfH. Möhring, and ChristosD. Zaroliagis, editors, *Robust and Online Large-Scale Optimization*, volume 5868 of *Lecture Notes in Computer Science*, pages 207–230. Springer Berlin Heidelberg, 2009.

[EFS11] Jochen Eisner, Stefan Funke, and Sabine Storandt. Optimal route planning for electric vehicles in large networks. In *AAAI*, 2011.

[Gei08] Robert Geisberger. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. Diplomarbeit, 2008.

[GH05] Andrew V. Goldberg and Chris Harrelson. Computing the shortest path: A search meets graph theory. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '05, pages 156–165, Philadelphia, PA, USA, 2005. Society for Industrial and Applied Mathematics.

[GP14] Michael T. Goodrich and Pawel Pszona. Two-phase bicriterion search for finding fast and efficient electric vehicle routes. *CoRR*, abs/1409.3192, 2014.

[GSSD08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In CatherineC. McGeoch, editor, *Experimental Algorithms*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer Berlin Heidelberg, 2008.

[HF14] Frederik Hartmann and Stefan Funke. Energy-efficient routing: Taking speed into account. In Carsten Lutz and Michael Thielscher, editors, *KI 2014: Advances in Artificial Intelligence*, volume 8736 of *Lecture Notes in Computer Science*, pages 86–97. Springer International Publishing, 2014.

[HNR68a] P.E. Hart, N.J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, July 1968.

[HNR68b] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *Systems Science and Cybernetics, IEEE Transactions on*, 4(2):100–107, 1968.

[HRZL09] Stefan Hausberger, Martin Rexeis, Michael Zallinger, and Raphael Luz. Emission factors from the model phem for the hbefa version 3. *Report Nr. I-20/2009 Haus-Em*, 33(08):679, 2009.

[II87] Hiroshi Imai and Masao Iri. An optimal algorithm for approximating a piecewise linear function. *Journal of information processing*, 9(3):159–162, jan 1987.

[Jaf84] Jeffrey M. Jaffe. Algorithms for finding paths with multiple constraints. *Networks*, 14(1):95–116, 1984.

[Joh73] Donald B. Johnson. A note on dijkstra's shortest path algorithm. *J. ACM*, 20(3):385–388, July 1973.

[KS93] David E. Kaufman and Robert L. Smith. Fastest paths in time-dependent networks for intelligent vehicle-highway systems application. *I V H S Journal*, 1(1):1–11, 1993.

[LL04] James Larminie and John Lowry. *Electric Vehicle Modelling*, pages 183–212. John Wiley & Sons, Ltd, 2004.

[MD79] R Garey Michael and S Johnson David. Computers and intractability: A guide to the theory of np-completeness. *WH Freeman & Co., San Francisco*, 1979.

[Mil12] Nikola Milosavljević. On optimal preprocessing for contraction hierarchies. In *Proceedings of the 5th ACM SIGSPATIAL International Workshop on Computational Transportation Science*, IWCTS '12, pages 33–38, New York, NY, USA, 2012. ACM.

[OR90] Ariel Orda and Raphael Rom. Shortest-path and minimum-delay algorithms in networks with time-dependent edge-length. *J. ACM*, 37(3):607–625, July 1990.

[SH76] Michael Ian Shamos and Dan Hoey. Geometric intersection problems. In *Foundations of Computer Science, 1976., 17th Annual Symposium on*, pages 208–215. IEEE, 1976.

[Sto12] Sabine Storandt. Quick and energy-efficient routes: Computing constrained shortest paths for electric vehicles. In *Proceedings of the 5th ACM SIGSPATIAL International Workshop on Computational Transportation Science*, IWCTS '12, pages 20–25, New York, NY, USA, 2012. ACM.

[VW93] Maheswari Visvalingam and JD Whyatt. Line generalisation by repeated elimination of points. *The Cartographic Journal*, 30(1):46–51, 1993.

[Zün14] Tobias Zündorf. Route planning for electric vehicles with realistic charging models. Master's thesis, 2014.

Appendix

A. Appendix

A.1. Numerical Issues

The integer function representation we utilize does not come without numerical problems, even though it avoids most of the typical such problems. We now present issues which break our implementations, instead of simply causing deviating results.

Merge Rounding. We start with a mild and expected issue. Whenever we merge two tradeoff functions, we must test whether two segments intersect. For this, we compute a line intersection and test whether it lies on both segments. At times, an inner intersection is in close proximity to one of the segment end-points and due to floating point imprecision lies outside one segment. This is a false negative which causes the merge algorithm (Chapter 3) to ignore an intersection. As a result, an entire section of the dominating function directly after the intersection is ignored. An improvement is therefore skipped, leading to an incorrect result. To avoid this problem we only compute intersections of segments with intersecting bounding boxes. We then test if either the intersection point or the rounded intersection point lies on the segments. An improvement would be to use the half-plane test, which would allow us to check whether two segments intersect without error.

No Termination. A more subtle problem causes an endless loop in the standard profile search (Chapter 4). The loop results from improvement propagation combined with the fact that we round intersections up. Consider vertices u and v with staircase-like profiles f_u resp. f_v, where f_u and f_v intersect at some points. Suppose the first half (w.r.t. τ axis) of f_u is computed correctly and in the second half rounding occurred. Suppose the opposite for f_v. A u–v path lands on the correct portion of f_u over the incorrect one of f_v. The same is done by a v–u path and the correct portion of f_v resp. incorrect portion of f_u. Given choice profiles on both paths, each time an improvement is propagated from v to u, and then back from u to v, the correct and incorrect portions of the two tradeoff functions swap. We solve this problem by testing, after each merge, whether the original function at the vertex dominates the merge result transposed by the point $(1, 1)$. If so, we ignore the improvement. Of course, this introduces further error to the algorithm. Note that this error is also included in Table 6.8.

Rounding Lower Bounds. As mentioned, during the CH preprocessing we round lower bounds down. Whenever we do so during the half-convex linking operation, we may compute incorrect results. Specifically, when merging two adjacent links ℓ_1 and ℓ_2 the merge stops at the first found intersection. If ℓ_2 has a portion which is rounded down, it is possible for ℓ_1 to intersect this portion. The linking of ℓ_1 will stop at this intersection and miss a portion of ℓ_1 which is contained in $\ell_1 \cup \ell_2$. This problem results in cases where the lower bound of a shortcut does not dominate the upper bound of the same shortcut. A possible solution to this problem is shifting the lower bound down by $(0, -1)$, whenever this occurs. However, we did not experience largely deviating results during the CH queries and so chose to omit such shifts.

Domination Tests. And last, whenever we check for domination we must multiply numbers. As soon as we start using function simplification, the gaps between interpolation points become very wide. At times, the multiplication results may no longer be contained by standard 32 bit integers and require larger containers (such as longs). When ignored the problem leads to computing wrong bounds: the domination test may give a false positive, causing the algorithm to skip function improvements. The latter steps of the CH query (Chapter 5) in turn become incorrect.