Karlsruhe Institute of Technology

Speed–Consumption Trade-Off for Electric Vehicle Routing

Bachelor Thesis of

Lorenz Hübschle-Schneider

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.-Inf. Julian Dibbelt
	Dipl.-Inf. Thomas Pajor

Time Period: 01.03.2013 – 30.06.2013

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, 28th June 2013

Abstract

In the area of route planning for battery-electric vehicles, previous works have focused on finding routes that are optimal regarding energy consumption and/or travel time, only considering a fixed speed level for each road segment, and thus ignoring the staggering energy consumption savings potential attainable through variation of driving speed. While computationally more tractable, the resulting routes often do not meet drivers' expectations, as for instance reducing speed on the traditionally fastest connection can be preferable to alternative routes that are more economical at maximum speed. In this thesis, we extend methods for solving multi-criterion path problems to take into account different speed levels on each road segment, e.g. considering discrete steps from 90 to 130 km/h on each part of a motorway. Furthermore, we adapt known techniques for speeding up computation, and develop new heuristic approaches to achieve further substantial improvements in running time. Experimental results on the road network of Western Europe show that our methods compute results with a very high degree of accuracy in reasonably practicable time, and underline the importance of speed variation to the trade-off between short travel times and low energy consumption.

Deutsche Zusammenfassung

Bisherige Arbeiten auf dem Gebiet der Routenplanung für batterieelektrische Fahrzeuge konzentrieren sich darauf, Routen zu finden, die bezüglich des Energieverbrauchs und/oder der Reisezeit optimal sind, nahmen dabei aber stets fixe Geschwindigkeiten auf jedem Straßenstück an. Damit wurde das enorme Potential bezüglich Verbrauchseinsparungen durch Geschwindigkeitsreduktionen ignoriert oder verkannt. Obgleich algorithmisch einfacher umsetzbar, erfüllen die von diesen Methoden gefundenen Routen oft nicht die Erwartungen von Fahrern, da beispielsweise eine langsamere Fahrgeschwindigkeit auf der klassisch schnellsten Verbindung Alternativrouten mit bei Höchstgeschwindigkeit niedrigerem Energieverbrauch vorzuziehen sein kann. In dieser Arbeit erweitern wir Methoden der multikriteriellen Routenplanung derart, dass sie auf jedem Straßenstück verschiedene Fahrgeschwindigkeiten in Betracht ziehen, beispielsweise in diskreten Schritten zwischen 90 und 130 km/h auf jedem Teilstück einer Autobahn. Zudem übertragen aus der Literatur bekannte Techniken zur Beschleunigung der Berechnung, und entwickeln neuartige heuristische Ansätze zur weiteren Laufzeitreduktion. Experimente auf dem Straßennetzwerk Westeuropas belegen, dass unsere Methoden in praktisch durchführbarer Zeit Ergebnisse von sehr hoher Genauigkeit berechnen, und unterstreichen die Bedeutung der Berücksichtigung verschiedener Geschwindigkeiten bei der Abwägung zwischen kurzen Reisezeiten und niedrigem Energieverbrauch.

Contents

1. Introduction

Although electromobility has gained substantial momentum in industrial development and production, little research has been conducted into the energy saving potential through intelligent route planning, considering factors such as speed and terrain.

Route planning, i.e. finding the best route from one place to another for some notion of 'best', guided by substantial improvements upon the methods used, has significantly gained importance over the last decade. Complex and slow programmes for home computers, distributed on optical discs, have been replaced by ever-present mobile phone applications with access to real-time traffic information. Vehicles that compute and execute routes autonomously are appearing on the horizon. These applications that make our lives easier all depend on the availability of fast and reliable route planning solutions.

Great amounts of research have been conducted into this area, first into static route planning, later into time-dependent scenarios involving historical and real-time traffic information. They share one trait, though, and that is a focus on conventional vehicles using internal combustion engines. The recent rise in popularity of electric vehicles demands new solutions, as the premises have changed: battery capacity and thus cruising range are severely limited, while driving down-hill and breaking allow for recuperation of energy. Charging is a time-intensive process and therefore not viable en route. It is clear that traditional route planning techniques do not suffice here, and new approaches are required.

Related Work

An overview of the field of route planning and speed-up techniques is given by Delling et al. [DSSW09]. The methods presented there can be classified into *goal-directed* approaches, directing the search towards the target by preferring edges that shorten the distance to it, and *hierarchical* ones, exploiting the topology of road networks. One example of a goal-directed technique is ALT [GH04], which uses landmarks with known distances to and from every point to provide estimates of distances to the target and favour roads that lead in the right direction. Another strategy is to label edges with flags indicating whether they might be useful when searching for a path to a specific destination region—this is known as the arc flags algorithm [HKMS06]. Hierarchical approaches often make use of contraction, where sequences of less relevant nodes are replaced by a shortcut, a principle used e.g. in Contraction Hierarchies [GSSD08]. Other hierarchical techniques, such as CRP [DGPW11], use multi-level overlays [DHM$^+$09, HSW08, JP02, SWW00, SWZ02] to limit the search to smaller graphs with increasing distance from the query's end points. In this thesis, we make use of the goal-directed *A** technique introduced by Hart, Nilsson and Rafael [HNR68] that utilises node potentials to pull the search towards the target. We have chosen A* search over the more complex, but also more efficient approaches, since it requires no additional pre-processing in our case, as a valid and in fact very good potential function is required at any rate (as explained in Section 4.2.1).

These speed-up techniques have subsequently been extended to handle multiple criteria, i.e. allow finding a set of routes of which none is outperformed by another in regard to all criteria, and thus all are optimal in some respect. An application of this is to reduce fuel consumption or avoid tolls without sacrificing too much travel time. To accomplish this, previous approaches needed to be generalised to utilise multi-dimensional node labels representing all Pareto-optimal paths [Han79, Mar84]. While theoretically hard, this problem is often feasible in practice [MW01], as new speed-up techniques have been developed [DMS08] and others have been adapted, such as A* search [SWI91] or SHARC [DW09], a combination of contraction and arc flags.

In the field of route planning for electric vehicles, most publications have focused on the integration of battery capacity constraints—the avoidance of overcharging and running out of energy—or negative edge weights—a result of recuperation—into classical single-criterion route planning algorithms [AHLS10, EFS11, SLAH11]. However, this is insufficient, as small detours can often yield significant energy savings, and drivers may expect a choice of routes, allowing them to pick one that suits their needs. Therefore, multi-criterion route planning techniques have been applied to the context of electric vehicles, computing Pareto-optimal paths regarding energy consumption and distance [Sto12]. Nevertheless, all of these approaches use a fixed speed level on each edge, not considering the potential savings from reducing speed to bring down energy consumption at the cost of some travel time.

In this thesis, we explore trading off speed and energy consumption to obtain a comprehensive algorithm for electric vehicle route planning. Our approach not only considers travel time and battery drain, but also explores possibilities of driving at different speeds. As a result, we obtain routes that drivers find desirable and convenient. Furthermore, we demonstrate that it is practically possible to compute such routes for today's electric vehicles on road networks of continental scale.

2. Preliminaries

This chapter lays the foundations for the thesis, introducing basic terminology used throughout the following chapters. The definitions given may therefore serve as a repetition of necessary background knowledge and to avoid possible ambiguities. The content of this chapter is arranged in two sections, first introducing basic definitions of graph theory, and then providing an overview of shortest path algorithms.

2.1 Graphs

A (weighted, directed) *graph* is a tuple $G = (V, E, c)$ of two finite sets V and $E \subseteq V \times V$ combined with a *weight function* $c : E \to \mathbb{R}$. The elements of V are called *nodes* or *vertices*, while E contains the *edges*. For any edge $(u, v) \in E$, we call $c((u, v))$ the weight, u the *head*, and v the *tail* of the edge. To increase readability, we define $c(u, v) = c((u, v))$. Two nodes u and v that are connected by an edge $e = (u, v)$ are also called *adjacent* or *neighbouring* and *incident* to the edge. The cardinalities of V and E are denoted by $n = |V|$ and $m = |E|$, respectively. If for each edge $(u, v) \in E$ the *backward edge* (v, u) is also contained in E, we call the graph *bi-directed.* If the graph also satisfies the condition $c(u, v) = c(v, u)$ for each edge $(u, v) \in E$, we call it *undirected.* Please note that this definition differs from most in literature in that we do not represent undirected edges as sets, i.e. $e = \{u, v\}$, but rather as two separate, directed edges (u, v) and (v, u) of the same weight.

Given a node v, we define its *in-degree* as $\text{in}(v) = |\{(u, v) \in E\}|$, while its *out-degree* is defined as $\text{out}(v) = |\{(v, w) \in E\}|$. The sum of these values is also called the node's *degree*: $\deg(v) = \text{in}(v) + \text{out}(v)$.

For two graphs $G = (V, E, c)$ and $G' = (V', E', c')$, we call G' a *subgraph* of G if and only if $V' \subseteq V$, $E' \subseteq E$, and for each edge $e \in E'$, $c'(e) = c(e)$.

A *path* in a Graph $G = (V, E, c)$ is a finite sequence of nodes $P = \langle v_1, \dots, v_n \rangle$ in V so that $(v_i, v_{i+1}) \in E$ for $1 \le i < n$. We write $P = \langle v_1, \dots, v_n \rangle \in G$. The *weight* or *length* $c(P)$ of P is defined as the sum of the weights of the edges corresponding to consecutive pairs of nodes on the path: $c(P) = c(v_1, v_2) + \cdots + c(v_{n-1}, v_n)$. For any such edge (v_i, v_{i+1}), we write $(v_i, v_{i+1}) \in P$. Should a path begin and end with the same node, i.e. $v_1 = v_n$, we call it a *cycle.* If no node v occurs more than once in the sequence of nodes, we call the path *simple.* Similarly, a *simple cycle* is a cycle where no node is part of the sequence more than once, except for v_1, which occurs a second time as v_n to close the cycle. For a path P, we say that $P = \langle v_1, \dots, v_n \rangle \in G$ if and only if all nodes on the path are nodes of the graph, i.e. $v_i \in V$ for all $i = 1 \dots n$, and for each of its consecutive pairs of nodes v_i, v_{i+1} with $1 \leq i < n$, there exists an edge $e = (v_i, v_{i+1}) \in E$.

Given a graph $G = (V, E, c)$, we define a *distance* function $d_G : V \times V \to \mathbb{R}$ on its nodes. If it is clear from context which graph is referred to, we omit the index. The function is defined as follows.

$$d_G(u,v) = \begin{cases} 0 & \text{if } u = v, \\ \infty & \text{if there is no path } \langle u, \dots, v \rangle \text{ in G}, \\ \min_{P = \langle u, \dots, v \rangle} c(P) & \text{otherwise.} \end{cases}$$

We call a graph $G = (V, E, c)$ *strongly connected* if for each pair of nodes $u, v \in V$, there exists a path $P_{u,v} = \langle u, \dots, v \rangle \in G$. A graph's *strongly connected components* are its maximal strongly connected subgraphs.

Given an undirected graph $T = (V_T, E_T, c_T)$, we refer to it as a *tree*, if and only if there is exactly one simple path connecting each disjoint pair of vertices. We define a *directed tree T rooted at s* as a directed graph $T = (V_T, E_T, c_T)$ with $s \in V_T$, $m = n - 1$, and for each vertex $v \in V_T$ there exists a simple path from s to v in T. If T is a subgraph of an arbitrary graph $G = (V, E, c)$, where V_T is the set of nodes reachable from s in G and $d_T(s, v) = d_G(s, v)$ for each node $v \in V_T$, we call T a *shortest-path tree* of s in G.

A *Directed Acyclic Graph* or *DAG* is a directed graph that does not contain cycles.

2.2 Shortest Paths

Shortest path problems typically fall into one of three categories: *Single-Pair Shortest Path Problems (SPSP)*, where we want to find the shortest path between a single pair of nodes, *Single-Source Shortest Path Problems (SSSP)* that require finding shortest paths from one node to all others, and the *All-Pairs Shortest Path Problem (APSP)*, where we compute the shortest path between any pair of nodes in the graph. In this work, we are interested in solving the Single-Pair Shortest Path Problem in the context of electric vehicles. On graphs with unidimensional weight functions, this problem can efficiently be solved with Dijkstra's algorithm.

2.2.1 Dijkstra's Algorithm

First described by Edsger W. Dijkstra in 1959 [Dij59], Dijkstra's algorithm is probably the best-known algorithm to solve the Single-Source Shortest Path Problem on weighted, directed graphs with non-negative edge weights. Listed in 2.1 is a variation of DIJKSTRA that not only computes shortest path distances, but also an implicit shortest-path tree. Its input is a Graph $G = (V, E, c)$ with a non-negative weight function $c : E \to \mathbb{R}_{\geq 0}$ and a source node $s \in V$. The algorithm's output consists of the shortest path distances from s, stored in the array $\mathsf{d}(\cdot)$, and a shortest-path tree rooted at s, implied by the parent pointers in the $\mathsf{pred}(\cdot)$ array. Both of these arrays have size n. To reconstruct the shortest path $P = \langle s = v_1, v_2, \dots, v_{n-1}, v_n = t \rangle$ from s to t, we follow the parent pointers from $t = v_n$ to v_{n-1}, from there to v_{n-2}, until $v_1 = s$ is reached. For any node v not reachable from s, $\mathsf{pred}(v)$ uses a special value, `null`, and $\mathsf{d}(v) = d(s, v) = \infty$. Therefore, a shortest-path tree $T = (V', E', c)$ rooted at s is given by $V' = \{v \in V \mid \mathsf{d}(v) < \infty\}$ and $E' = \{(\mathsf{pred}(v), v) \mid v \in V' \backslash \{s\}\}$.

Algorithm 2.1: Dijkstra

```
Input: Graph $G = (V, E, c)$, source node $s$
Data: Priority queue Q
Output: Distances $\mathsf{d}(v)$ for all $v \in V$, shortest-path tree of $s$ given by $\mathsf{pred}(\cdot)$
// Initialisation
forall $v \in V$ do
    $\mathsf{d}(v) \leftarrow \infty$
    $\mathsf{pred}(v) \leftarrow$ null
Q.update$(s, 0)$
$\mathsf{d}(s) \leftarrow 0$
// Main loop
while Q is not empty do
    $u \leftarrow$ Q.deleteMin()
    forall $(u, v) \in E$ do
        if $\mathsf{d}(u) + c(u, v) < \mathsf{d}(v)$ then
            $\mathsf{d}(v) \leftarrow \mathsf{d}(u) + c(u, v)$
            $\mathsf{pred}(v) \leftarrow u$
            Q.update$(v, \mathsf{d}(v))$
```

Description of the Algorithm

The procedure, shown in Algorithm 2.1, makes use of a *priority queue* data structure that manages objects paired with integer keys. While not giving a specific implementation, we require it to provide the following operations:

- update(v, k), which either inserts node v with key k if v was not in the queue, or *decreases* its key to k if it was.
- deleteMin(), which returns the node with *minimum* key and removes it from the queue.

In the beginning, the algorithm initialises its data structures by setting all distances to infinity, except for the starting node, which has distance 0, and storing the special value `null` as parent pointer for each vertex. Until the algorithm finishes, we refer to the entries of $\mathsf{d}(\cdot)$ as *tentative distances*. In the beginning, it inserts the start vertex into the queue with key 0. At that point, the main loop starts and terminates only when the last node has been removed from the queue. In each step, the node u with minimal key is retrieved and removed from the queue. We now call u *settled*. Next, we apply a process called *scan* to each outgoing edge of u. When scanning an edge (u, v) in line 9, we check if the shortest path $P_u = \langle s, \ldots, u\rangle$ from s to u plus the edge (u, v), $P_v = \langle s, \ldots, u, v\rangle$, is shorter than the current, tentative shortest path from s to v. If this is the case, we *relax* the edge in lines 10–12 by setting the tentative distance of v, $\mathsf{d}(v)$, to the weight of the new path: $\mathsf{d}(v) = \mathsf{d}(u) + c(u, v)$. Since the current shortest path to v now passes through u, we need to update the parent pointer as well and set $\mathsf{pred}(v) = u$. Lastly, we need to set the new key of v in the priority queue. This ends the process of relaxation and scanning.

Correctness The correctness of Dijkstra's algorithm follows from the facts that every reachable node is settled, $\mathsf{d}(v)$ is always greater or equal to $d(s, v)$, and that once a node v is settled, $d(s, v) = \mathsf{d}(v)$.

The first claim can be verified easily. Assume that there is a node v that is reachable from s but not settled by the algorithm. Then there must exist some path P from s to v, i.e. $P = \langle s = v_1, v_2, \dots, v_{n-1}, v_n = v\rangle$. We also know that s is settled by the algorithm in the very first step. Therefore, there must be some node $v_i \in P$ so that its predecessor on the path, v_{i-1}, is settled, but not v_i. However, since v_{i-1} is settled and adjacent to v_i, v_i is added to the queue as well, and therefore v is also settled by the algorithm. This is a contradiction.

To prove the second claim, recall that initially each node v has its tentative distance $\mathsf{d}(v)$ set to ∞. All changes to tentative distances correspond to lengths of paths from s to v in G. Since each path from s to v is at least as long as the shortest path, the claim is true.

The last claim is correct because once a vertex v is removed from the queue, all remaining nodes have greater or equal key. Since no edge weights are negative, the queue's minimum key only ever increases during execution of the algorithm. Therefore, any node u still in the queue when v is removed cannot be on the shortest path from s to v, and neither can any node that has not yet been inserted into the queue as by the first claim. Thus the shortest path has already been found when v is extracted from the queue. We call this the *label-setting property.*

Stopping Criterion If our interest is merely in finding the shortest path to a single target node t, i.e. solving the Single-Pair Shortest Path Problem, we can introduce a stopping criterion to the search. Recalling the label-setting property, it can easily be seen that we can halt the algorithm once the target node has been removed from the queue. Although this does not result in an improvement of asymptotic complexity, it can reduce practical running time significantly, especially for local queries.

Complexity Label-setting DIJKSTRA performs at most n DELETEMIN operations on the queue—one for each vertex—and no more than m UPDATE operations, as each edge is scanned at most once. When implemented with a Fibonacci heap data structure as priority queue, this results in a total running time complexity of $\mathcal{O}(m + n \cdot \log n)$ [FT87].

Negative Edge Weights and Label-Correcting Search

Dijkstra's algorithm as introduced above has the limitation of requiring edge weights to be non-negative. While this condition is the key to the label-setting property, we may want to allow negative edge weights, as long as *negative cycles*, i.e. cycles with negative weight, do not occur. For an example, imagine finding the most energy-efficient route for an electric vehicle in a road network. Due to recuperation, downhill roads may have negative energy consumption and thus correspond to edges with negative weight. Additionally, in this model, negative cycles cannot occur—they would allow us to recharge the battery arbitrarily by repeatedly driving a round-trip and thus allow construction of a perpetual motion machine of the first kind. Thankfully, adapting DIJKSTRA to this case is simple—in fact, Algorithm 2.1 is already fully equipped to handle negative weights in graphs without negative cycles. This is the case because we use the UPDATE function in line 12, which can re-insert nodes into the queue even if they have been previously removed from it. The possibility to do so is required when reaching a node via an edge with negative weight, thus lowering its tentative distance *after* it has already been settled, and thus reactivating it. Therefore, Dijkstra's algorithm with negative edge weights does not possess the label-setting property and is called *label-correcting.* The ability to settle vertices multiple times also requires us to re-evaluate the algorithm's running time complexity, which is now exponential in the worst case [Joh73].

Potential Shifting To avoid the increased complexity resulting from the loss of the label-setting property on graphs with negative edge weights, we may also employ a technique called *potential shifting*, first introduced by Johnson [Joh77]. It calculates a potential function for the graph's nodes that is used to augment the edges' weights, resulting in a re-weighted graph without negative edge weights. The shortest path between any pair of nodes of this new graph uses the same sequence of edges as the shortest path between the same nodes of the original graph. To compute the potential function, we add a new vertex s to the graph and connect it to each other vertex by zero-weight edges. We then run a label-correcting search starting from this new vertex s, either using the above algorithm or the Bellman-Ford algorithm introduced by Bellman in 1958 [Bel58], and obtain shortest path distances from s in an array $\mathsf{d}(\cdot)$. Next, we re-weight the edges, assigning each edge $e = (u, v)$ its new weight $c'(u, v) = c(u, v) + \mathsf{d}(u) - \mathsf{d}(v)$. Lastly, s and all edges incident to it are removed from the graph, restoring its original structure. As all new edge weights are non-negative, Dijkstra's algorithm can be run on the re-weighted graph, followed by a reconstruction step to compute the real weight of the shortest path. Alternatively to potential shifting, we can employ a technique called *A* search*, which works similarly but is considerably faster when searching for the shortest path to a single target.

A* Search

With a reasonable choice of priority queue data structure, Dijkstra's algorithm solves the SSSP problem efficiently. If we are only interested in the shortest path towards a single target node t, however, we can employ a wide range of speed-up techniques that all yield provably correct results for the SPSP problem and decrease measured running times, despite not improving asymptotic behaviour. Here, we explore the A* search algorithm, which speeds up the search by figuratively "pulling" it towards the target. To achieve this, we introduce a *potential function* $\pi_t : V \to \mathbb{R}$ that is added to the nodes' keys in the queue, i.e. a node v with key $\mathsf{d}(v)$ is now assigned key $\mathsf{d}(v) + \pi_t(v)$. To maintain correctness, we have to impose some limitations on what is considered a valid potential function. We call a potential function $\pi_t : V \to \mathbb{R}$ *feasible* with regard to a graph $G = (V, E, c)$, if and only if for each edge $(u, v) \in E$ the inequation $\overline{c}(u, v) := c(u, v) - \pi_t(u) + \pi_t(v) \geq 0$ holds. We then call $\overline{c} : E \to \mathbb{R}_{\geq 0}$ the *reduced* weight function. If we additionally require the target's potential to be zero, we can think of $\pi_t(v)$ as a lower bound on the distance from v to t. Again, we can stop the search when t has been removed from the queue [HNR68]. Note that A* search is always label-setting, as reduced weights are non-negative, and thus obviates the need for Johnson's potential shifting technique detailed above. Additionally considering the speed-up it provides by favouring edges that reduce the distance to the target, it is often the better choice when solving the SPSP problem.

2.2.2 Multi-Criterion Search

Dijkstra's algorithm works well when computing shortest paths regarding a one-dimensional weight function like travel time. When extending the graph's weight function c to the multi-criterion case $c : E \to \mathbb{R}^n_{\geq 0}$, i.e. replacing the edges' scalar weights with vectors of non-negative real numbers, it is not clear anymore which path is to be considered the best one. Now, instead of a single path, our objective is to find a set of paths that are all *non-dominating*. We call a path *dominated* if there exists another path in the solution which is better with regard to *all* criteria. The set of all non-dominated paths shall be called the *Pareto set* and its elements the *Pareto-optima* hereinafter. To extend Dijkstra's algorithm to compute the Pareto set, we have to replace the single label associated with each vertex with a *bag* of labels, storing the node's (tentative) Pareto set. Each of these labels contains what was previously stored in the single label: the value for each criterion and, optionally, a parent pointer to the vertex from which the node was reached when creating the label. Note that to retrace a path, we need to store parent pointers to labels and not bags, since a bag may contain labels stemming from different paths. The bag

Algorithm 2.2: Multi-Criterion Search

```
Input: Graph $G = (V, E, c)$ with $c : E \to \mathbb{R}^n_{\geq 0}$, source node $s$
Data: Priority queue Q
Output: Pareto set of distances for all nodes, multi-criterion shortest-path tree of $s$

// Initialisation
1  forall $v \in V$ do
2      bag($v$) ← $\emptyset$
3  bag($v$) ← LABEL($0,\dots,0$)
4  Q.UPDATE($s$)
// Main loop
5  while Q is not empty do
6      $u$, bag ← Q.FRONTELEMENT()
7      currentLabel ← bag.POPFIRSTUNPROCESSEDLABEL()
8      if bag.HASUNPROCESSEDLABEL() then
9          Q.EXTRACTFRONT()
10     else
11         Q.UPDATE(bag)
12     forall $(u, v) \in E$ do
13         newHeadLabel ← currentLabel + $c(u, v)$
14         if BAG($v$).MERGELABEL(newHeadLabel) then
15             Q.UPDATE(BAG($v$))
```

itself is sorted according to an ordering we can choose freely. Commonly, a lexicographical ordering with respect to the criteria is used.

Dijkstra's algorithm needs to be changed in a few regards, as can be seen in Algorithm 2.2. Instead of vertices, the priority queue now needs to manage bags. A bag's *key* is its first label according to the ordering discussed above. We require the bag data structure to provide the following operations:

- hasUnprocessedLabel(), which checks if the bag contains an unprocessed label,
- popFirstUnprocessedLabel(), which returns the first unprocessed label and marks it as processed,
- mergeLabel(label), which merges label into the set of labels if and only if it is not dominated by any existing label, removes all existing labels that are now dominated, and re-orders the labels to maintain consistency.

In each iteration of the main loop, the bag with the lowest key is chosen, and its first unprocessed label is inspected and marked as processed. If the bag has no further unprocessed labels, we can remove it from the queue, otherwise we perform an update operation to inform the queue of the bag's new key. Then, all of the node's outgoing edges are scanned with this label. When doing so, we generate a new label for each edge just as we would do in the single-criterion case. Next, we need to check if this new label is *dominated* by the bag, i.e. if there is another label already in the bag that dominates our new one. If this is the case, we abort the scan. Otherwise, we *relax* the edge by *merging* the label into the bag, discarding all others that are now dominated. Thereby, we ensure that no two labels in the bag dominate each another at any time, i.e. each label in the bag represents a (tentative) Pareto-optimum. Lastly, we update the queue, inserting or updating the head vertices bag.

We can infer an interesting property from this procedure. Because in each iteration, the label with the globally lowest key is chosen for relaxation, and that edge weights are non-negative in each criterion, no already processed label is ever dominated by a newly inserted one. This is to multi-criterion search what the label-setting property is to Dijkstra's algorithm, and we thus call the procedure *multi-label-setting*.

Complexity

The running time complexity of the multi-criterion search algorithm depends on the global number of Pareto-optima. Hansen [Han79] has shown that this number can be exponential in the worst case. Even the *weight-restricted shortest path problem* [MZ00], a simplifying variation of the bicriterion case that optimises the first criterion under the condition that the second one does not exceed a certain value, is still NP-complete, as shown by Handler and Zang [HZ80]. However, the problem is often considerably less hard in practice if the criteria correlate well, as e.g. travel time and distance do. This makes the Pareto front's computation feasible in practice for many cases [MW01]. Unfortunately, electric vehicles' energy consumption, involving recuperation, can result in little correlation with travel times, increasing the number of Pareto-optimal solutions and thus running time.

3. Basic Approach

This chapter explains how the multi-criterion search procedure shown in Algorithm 2.2 of the Preliminaries Chapter needs to be extended to enable trading speed for energy consumption savings and thus range extension. In the first section, we introduce our extensions to the graph model and the necessary data structures. Secondly, Section 3.2 explores the simplified problem of optimising the speed–consumption trade-off for a single path. Section 3.3 then expounds the idea of our basic algorithm.

3.1 Modelling

In order to find the trade-off between travel time and energy consumption, we need to extend our graph model to include information about possible speed levels and the energy consumption entailed.

3.1.1 Input Data

In addition to a *road network*, which typically contains vertices, i.e. intersections, and edges, i.e. road segments connecting the vertices, we require information about speed levels and energy consumption. Therefore, we associate with each edge a *consumption table*, which maps driving speed to the time travelled and the energy consumed along this road segment at that particular speed level. Since our goal is to give useful instructions to drivers, it is suitable to use discrete speed levels and pre-compute energy consumption. For each of these driving speeds, the consumption table associated with an edge provides the travel time and energy consumption when driving along the road segment represented by the edge at that speed. Formally, a consumption table is a set of tuples $\{(t_1, c_1), \dots, (t_k, c_k)\}$ containing travel times t_i and their associated energy consumption values c_i $(1 \leq i \leq k)$. If reconstruction of driving instructions is necessary, the speed values need to be stored as well. An alternative way to model consumption tables is to use multi-edges, i.e. multiple edges that connect the same pair of vertices in the same direction, but have different cost. In our case, we would insert one edge for each speed level, associating with it a cost tuple (t, c) specifying the travel time and energy consumption at this level. Although equivalent, this formulation would be inefficient in implementation due to edge overhead and serves for exemplification only.

Limitations on Speed Variations

To avoid erratic speed changes, we introduce some limitations on when we allow a change in driving speed. For example, it is undesirable to frequently ac- and decelerate on stretches of roads. Therefore, we do not consider changing velocity at vertices where conditions remain unaltered, i.e. the driver does not see any change in the road. As these conditions rely on specifics of the input data used, we refer the reader to Section 5.1.1 of the Experiments Chapter for details. At this point, it suffices to say that we require the same set of speed levels to be driven on all pairs of edges between three distinct vertices—otherwise, we could exceed speed limits unwillingly, and we do not consider U-turns—, and assume that vertices where these conditions are met have a special flag set, which we will call the *compression* flag. As forward and backward edges are stored separately, it is also possible to compress a vertex in one direction only, which we mark appropriately.

Contraction of Nodes with Exactly Two Neighbours

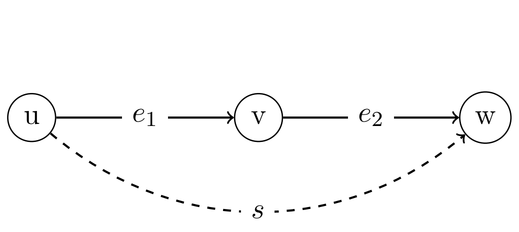
In route planning, it is common to contract nodes—recall that we use *vertex* and *node* as synonyms—that have exactly two neighbours, as these typically serve geometric modelling on a map. Removing these nodes and merging the edges incident to them simplifies routing by removing complexity from the graph. Usually, this amounts to adding a new edge with the sum of the costs of the removed edges and deleting the newly isolated vertex. We perform something similar, but instead of simply summing up weights, consumption tables have to be merged, and contraction is limited to nodes for which the compression flag is set. Instead of enforcing the speed variation limitations in our algorithm, we simply contract such vertices in the data aggregation step.

When contracting a node, we have to create *shortcut* edges to replace those incident to the node, which are removed with it. When doing so, we compute a new consumption table for the shortcut edge by summing up travel time and energy consumption. Recall that we require speed levels on the two edges which are to be merged to be the same. Formally, two edges $e_1 = (u, v)$ and $e_2 = (v, w)$ with lengths ℓ_1 and ℓ_2, and consumption tables $\{(t_{1,1}, c_{1,1}), \ldots, (t_{1,k}, c_{1,k})\}$ and $\{(t_{2,1}, c_{2,1}), \ldots, (t_{2,k}, c_{2,k})\}$, respectively, are merged into a shortcut edge $s = (u, w)$ with length $\ell = \ell_1 + \ell_2$. As speed values $s_{1,i}$ and $s_{2,i}$ are equal for all $1 \leq i \leq n$, we can contract the vertex and the consumption table of s is given by $\{(t_{1,1} + t_{2,1}, c_{1,1} + c_{2,1}), \ldots, (t_{1,k} + t_{2,k}, c_{1,k} + c_{2,k})\}$.

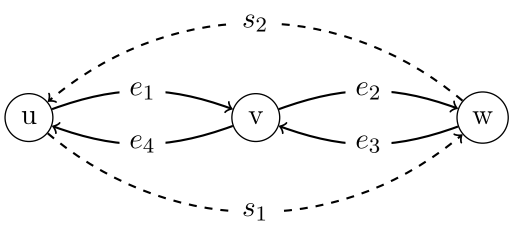
There are two kinds of nodes where such contraction is possible: for one, directed stretches where a node v has exactly two neighbours u, w so that $e_1 = (u, v) \in E$ and $e_2 = (v, w) \in E$, but there exist no other incoming or outgoing edges of v, as displayed in Figure 3.1(a). In this case, we insert a shortcut $s = (u, w)$ into the graph and delete e_1, e_2 and v. The stretch may also be bi-directed, so that apart from e_1 and e_2, their reverse edges $e_3 = (w, v), e_4 = (v, u) \in E$ are the only edges incident to v, see Figure 3.1(b). In that case, two shortcuts $s_1 = (u, w)$ and $s_2 = (w, u)$ need to be created, which generally have different consumption tables. Once both shortcuts have been inserted into the graph, we remove the edges e_1, e_2, e_3, e_4 they replace and delete the vertex v.

We remark that in the case of a vertex that is only compressible in one direction, it is not removed from the graph, and neither are the edges that correspond to the direction that is not compressible. A shortcut for the compressible direction is introduced nonetheless.

Observe that we do not perform any extensive pre-processing on the graph. While the contraction of nodes with two neighbours may be considered pre-processing, it is there to enforce our limitations on speed variation and not to pre-compute anything to later be used for speeding up execution of the main algorithm.



(a) Directed Stretch



(b) Bi-directed Stretch

Figure 3.1: Two examples for contraction of nodes with two neighbours. Dashed edges indicate the shortcuts inserted during contraction.

3.1.2 Data Structures

The graph and consumption tables serve as input to our algorithm. We now proceed to discuss data structures and operations needed during execution of the algorithm.

Bags and Labels

With each vertex we associate a *bag* which holds a set of *labels*. Each label contains the energy consumption value and travel time for the path that was taken to create it. Recall that, as introduced in Section 2.2.2, labels need to reference the *label* they originated from to allow for path reconstruction, as it no longer suffices to know only the *vertex* or bag from which it was spawned. To reconstruct driving instructions, we also need to store the speed that was driven on the road segment corresponding to the label. In a node's bag, we store all non-dominated labels leading to it, ordered ascendingly by energy consumption. For more on label domination and bag structure, please refer to Section 2.2.2 of the Preliminaries Chapter. We also need to remember which labels of each bag have already been processed and thus keep a pointer to its first unprocessed label. A bag's *key* is its first, i.e. most energy-saving, unprocessed label.

We observe than in the case of non-negative edge weights, already processed labels cannot be dominated, which is the equivalent of the label-setting property in Dijkstra's algorithm. Edge weights are non-negative if we either forbid recuperation of energy, skewing results heavily in favour of flat terrain, use Johnson's potential shifting technique from Section 2.2.1 on each of the criteria that can be negative [Joh77, EFS11], or A* search as detailed in Section 4.1.2.

Merging Labels When *merging* a new label into a bag, we first need to check whether any of the already existing labels dominates the candidate. If this is the case, we discard it immediately, as a dominated label cannot produce non-dominated results. Otherwise, we remove any labels from the bag that are dominated *by* the new label and insert it in the correct position to maintain the bag's ordering.

Priority Queue

To track vertices' bags and quickly determine which one to process next, we use a priority queue data structure of which we require the following operations to be implemented:

- FRONTELEMENT(), which returns the bag with the lowest key,
- EXTRACTFRONT(), which removes the bag with the lowest key from the queue, and
- UPDATE(BAG, π), which inserts the bag into the queue, or, if it has already been inserted, updates its position, *optionally* adding a (multi-dimensional) potential π to its key, which we later require when implementing A* search

We do not give a description or implementation of such a data structure, as they can be found easily in literature. A good overview of different types of priority queue implementations is given in [MS08].

3.2 Optimising a Single Path

We begin by restricting the problem to finding all viable combinations of speeds on a single path $P = \langle s = v_1, v_2 \ldots v_{n-1}, v_n = t \rangle$. Each edge can be traversed at different speeds, resulting in varying energy consumption. Our approach is to compute potential savings, defined as energy consumption reduction per additional driving time, including ac- and deceleration, for each edge. In every step, we choose the edge with maximum potential savings, reduce driving speed on it, compute the new potential savings for the edge, and update its neighbours' acceleration penalties. We call this a *reduction* of the edge. This can be implemented well with a priority queue that supports increasing keys in addition to decreasing them.

Optimality and Modelling Limitations In the graph model, we can only drive one speed on the whole of an edge, which may represent a rather long distance, while in the real world, drivers can change their driving speed much more flexibly. As we are only optimising the speeds on a single path and do not have the ability to deviate from it, we take the liberty of diverging from the edge model to compute results that are better than what would be considered optimal in the fixed-speed-per-edge approach. The result is that we compute the lower convex hull of the path's Pareto front in the graph model. When we want driving instruction for some goal in between two Pareto points, we simply choose the Pareto point that overachieves our goal by the least amount and reduce speed on a *part* of the last edge that was reduced. Thereby, we can achieve better results than if we simply calculated the Pareto front of the path in the graph model, and do so faster.

Implementation To implement single-path optimisation, we need to introduce a new element data structure for use with our priority queue, which we previously introduced in Section 3.1.2. We assume that each element knows its energy consumption cons, its travel time time, as well as its neighbours pred and succ, and require it to provide the following operations:

- hasSmallerKey(other) to compare elements in the queue,
- canReduce(), which checks whether a further reduction step is available,
- reduce(), which executes one reduction step and updates its state as well as its neighbours',
- penalty(), which returns a tuple of penalties in travel time and energy consumption incurred by the next reduction step. If no reduction step is possible, behaviour is undefined. Negative values indicate savings.

Additionally, we make use of the Bag data structure introduced in Section 3.1.2. Pseudocode for the single-path optimisation algorithm is shown in Algorithm 3.1.

Theorem 3.1. *The procedure listed in Algorithm 3.1 computes the lower convex hull of the graph model's Pareto front*

Proof. We conduct the proof in two steps: First, we show that all result points are Pareto points. In the second half, we show that the subset of Pareto points found by this procedure forms the lower convex hull of the Pareto front.

Algorithm 3.1: Single-Path Optimisation

```
Input: A set of road segments as priority queue elements elems
Data: A priority Queue Q
Output: A Bag bag containing labels corresponding to optimal solutions
// Initialisation
1  (cons, time) ← (0, 0)
2  forall elem ∈ elems do
3      Q.update(elem)
4      (time, cons) ← (time + elem.time, cons + elem.cons)
5  while Q is not empty do
6      elem ← Q.frontElement()
7      (time, cons) ← (time, cons) + elem.penalty()
8      elem.reduce()
9      bag.mergeLabel(Label(time, cons))
       // Update Queue
10     if elem.canReduce() then
11         Q.update(elem)
12     else
13         Q.extractFront()
14     Q.update(elem.pred)
15     Q.update(elem.succ)
```

Part 1. We denote the result sequence as $R = \langle r_1, \dots, r_n \rangle$ and the Pareto front as $P = \langle p_1, \dots, p_k \rangle$. Both are sorted according to the bags' ordering, i.e. lexically with respect to energy consumption and travel time. Assuming that some point $r_i = (t, c) \in R$ is *not* part of the Pareto front, i.e. $r_i \notin P$, there must exist a point $p_j \in P$ that dominates r_i, as all result points correspond to valid journeys. However, to dominate r_i, $p_j = (t', c')$ has to fulfil the conditions $t' \leq t$, $c' \leq c$, and $(t, c) \neq (t', c')$. Visually speaking, this means that on a plot of the Pareto front, where the abscissa specifies travel time and the ordinate corresponds to energy consumption values, p_j has to be to the left and/or below r_i. However, in each step our procedure chooses the reduction that provides the most energy consumption savings per time cost. In the above plot, this would be the *steepest* of the possible reductions. Therefore, the reductions chosen by our procedure cannot be dominated by any point of the Pareto front.

Part 2. If a Pareto point $p_i = (t_i, c_i)$ is not on the lower convex hull, and its neighbours are $p_{i-1} = (t_{i-1}, c_{i-1})$ and $p_{i+1} = (t_{i+1}, c_{i+1})$, then the quotient of the consumption savings and the travel time increase is less for the step from p_{i-1} to p_i than for the step from p_i to p_{i+1}:

$$\frac{c_{i-1} - c_i}{t_i - t_{i-1}} < \frac{c_i - c_{i+1}}{t_{i+1} - t_i}$$

In the plot described in the first part of the proof, this means that the first reduction step is less steep than the second. However, as in each step, the steepest reduction is chosen, and p_i is not the result of such a step, it cannot be part of the result sequence R. □

Complexity The number of priority queue operations is strictly limited, as each of the k edges has a fixed number of possible reductions, of which we will call the maximum ℓ. If we do not abort the above procedure, it will reduce each edge to each level at some point, yielding a total of $\mathcal{O}(k \cdot \ell)$ reductions. Each of these consists of four priority queue operations—one deleteMax to retrieve the edge from the queue and three update calls, one for the edge and one for each of its neighbours—and constant potential computation complexity. When using a binary heap, this amounts to a running time of $\mathcal{O}(k \cdot \ell \cdot \log(k \cdot \ell))$. Note that this requires each road segment's subsequent reductions to have *lower* potential savings than the current one. We call this the *monotonicity condition*, where monotonicity refers to the derivative of the potential savings function.

Complications in Non-Monotonous Scenarios

If the above condition of non-monotonicity does not hold, our simple priority queue concept is no longer directly applicable, potentially making the problem significantly harder. As a possible solution, we suggest extending the structure of the priority queue elements in a way that allows us to continue using the greedy approach. In the simple scenario, we can imagine the elements as items in a linked list of possible follow-up levels. Upon reduction, the element is removed from the queue while inserting its successor. When transferring this idea on to the scenario with non-monotone potential savings, we need to introduce the concept of shortcuts. A shortcut in this case, much like it is the case with shortcut edges in a graph, comprises multiple reduction steps summarised in a single one. Therefore, reducing along a shortcut is equivalent to multiple consecutive ordinary reduction steps. This allows us to create shortcuts spanning non-monotone reduction steps, but it also means that we no longer can use the analogy of a linked list but instead must replace it with a Directed Acyclic Graph (DAG): multiple successor elements are a possibility (requiring a tree), but elements may share successors to eliminate redundancy, thus requiring a DAG. We also have to introduce a slight limitation—to obtain correct results, the *last* reduction step performed must not be a shortcut, as partially taking multiple steps at once can yield incorrect results.

Analysis To understand how this will impact the number of priority queue operations, we need to determine the structure of this DAG. It is clear that whatever follows the last non-monotone element need not be changed in any way. For the elements that precede it, however, we need to add a shortcut wherever the potential of multiple reduction steps at once exceeds that of the currently topmost one. In the worst case, the number of elements in this DAG is exponential in the level of the last non-monotone element. However, at any given point, we only keep the children of the element reduced last in the priority queue, which is at most the number of successive non-monotone elements. If we generate our elements lazily and not up-front, the increase in priority queue operations is minor given the rather small number of total reduction steps per road segment, which usually is in the single digits, let alone that of consecutive non-monotone reduction steps and the number of edges affected by this phenomenon. In the absolute worst case that each of the k edges on the route has ℓ reduction steps, each of which yields better potential savings per additional time spent than the previous, we need to insert (and remove) $\mathcal{O}(\ell)$ elements into (or from) the priority queue for each of the ℓ reduction steps, resulting in at most $\mathcal{O}(k \cdot \ell^2)$ priority queue operations, compared to $\mathcal{O}(k \cdot \ell)$ operations in the simple case.

3.3 Extending Multi-Criterion Search

We new present the changes to multi-criterion search as it was introduced in Section 2.2.2 and listed in Algorithm 2.2, upon which our approach is based. In the following, we describe the modifications to be made for handling multiple speed levels. Most notably, we extend edge scans to spawn a new label for each entry of the edge's consumption table. Since edges typically have two to five consumption table entries, the potential number of non-dominated labels increases sharply.

First, we recall the basic principle of multi-criterion search. The overall approach is very similar to Dijkstra's algorithm, however, we have to replace the vertex labels with bags containing a number of labels. This is due to the fact that when considering multiple criteria, a total ordering does not exist—we cannot say that a label with a low value in one criterion and a high one in another is strictly better than a label with a higher value in the first criterion and a lower one in the second. Thus, we keep all labels that provide a non-dominated point on the Pareto curve. Remember that a label *dominates* another one only if it is better with respect to *all* criteria.

The main difference between our approach and the basic multi-criterion search algorithm is that during an edge scan, the edge's consumption table needs to be consulted. For each speed level, we retrieve travel time and energy consumption along the edge from the consumption table and add these values to the current label to obtain the head label. We then check that the energy consumption does not exceed the battery's capacity in either direction, depleting or overcharging it. If the battery capacity is insufficient, the edge scan continues with the next speed level. In the case of overcharging, we cap recuperation at a full charge. Once energy consumption is within the limits, we try to merge the label into the head vertices bag and proceed with the next speed level. After the consumption table has been iterated completely, we update the queue to reflect the changes to the head vertices bag. Then, the next outgoing edge is chosen for scanning, and once there is no further one, the label has been processed successfully and we proceed with the main loop's next iteration.

For the sake of simplicity, we assume the battery to be fully charged in the beginning. This limitation can easily be lifted by adapting the start vertices initial label to reflect the battery capacity missing to a full charge.

Effects of Recuperation As our model allows for recuperation of energy, e.g. when driving down-hill, we need to examine the effects of negative energy consumption on the number of labels created. Therefore, we lift the restriction that the weight function be non-negative in all components. We have already seen that if no negative values occur, a label that has already been processed can never be dominated by a subsequently merged label. This is due to the same insight that the label-setting property of Dijkstra's algorithm is based on. Equally, negative energy consumption allows us to dominate labels that have already been processed. This may result in a very high number of labels being created. To avoid this, a potential shifting technique could be applied to the energy consumption values, e.g. Johnson's algorithm, which we introduced in Section 2.2.1. We show a different approach to edge re-weighting that is based on A* search and does not require pre-processing, but instead runs in the initialisation phase of the algorithm, in Section 4.1.2 of the Optimisations Chapter. Our experiments on real-world data showed a nearly thirty-fold increase in running time on the same set of queries if edges were not re-weighted. For details, please refer to Section 5.2 of the Experiments Chapter.

The basic, unoptimised approach, implemented in pseudocode, can be seen in Algorithm 3.2.

Correctness

We now show that the procedure presented in Algorithm 3.2 fulfils the claims made about its results.

Theorem 3.2. *Algorithm 3.2 correctly calculates the Pareto set and paths from the source vertex to all other vertices in the graph, where energy consumption on no path exceeds the battery capacity specified.*

Algorithm 3.2: Our basic algorithm

```
Input: Graph $G = (V, E, \textsc{cons}, \textsc{time})$ with $\textsc{cons}, \textsc{time} : E \to \mathbb{R} \cup \cdots \cup \mathbb{R}^\ell$ as
       consumption tables, source node $s$, battery capacity $\mathsf{maxCons}$
Data: Priority queue $\mathsf{Q}$, a bag $\textsc{Bag}(v)$ for each vertex $v \in V$
Output: Pareto set and paths from the source to each vertex under the constraint
        that no more than $\mathsf{maxCons}$ energy be used

   // Initialisation
1  forall $v \in V$ do
2      $\textsc{Bag}(v) \leftarrow \emptyset$
3  $\textsc{Bag}(s) \leftarrow \textsc{label}(0,0)$
4  $\mathsf{Q}.\textsc{update}(\textsc{Bag}(s))$
   // Main loop
5  while $\mathsf{Q}$ is not empty do
6      $u$, $\mathsf{bag} \leftarrow \mathsf{Q}.\textsc{frontElement}()$
7      $\mathsf{currentLabel} \leftarrow \mathsf{bag}.\textsc{popFirstUnprocessedLabel}()$
8      if $\mathsf{bag}.\textsc{hasUnprocessedLabel}()$ then
9          $\mathsf{Q}.\textsc{extractFront}()$
10     else
11         $\mathsf{Q}.\textsc{update}(\mathsf{bag})$
12     forall $(u, v) \in E$ do
           // Iterate over the speed levels
13         forall levels $n \leftarrow 1, \ldots, k \leq \ell$ do
14             $\mathsf{newHeadLabel} \leftarrow \mathsf{currentLabel} + (\textsc{cons}(n), \textsc{time}(n))$
15             $\mathsf{newHeadLabel}.\textsc{cons} \leftarrow \textsc{max}(0, \mathsf{newHeadLabel}.\textsc{cons})$
16             if $\mathsf{newHeadLabel}.\textsc{cons} > \mathsf{maxCons}$ then
                   // Battery charge depleted
17                 continue
18             $\mathsf{newHeadLabel.previousLabel} \leftarrow \mathsf{currentLabel}$
19             $\textsc{Bag}(v).\textsc{mergeLabel}(\mathsf{newHeadLabel})$
20         if $\textsc{Bag}(v)$ was modified then
21             $\mathsf{Q}.\textsc{update}(\textsc{Bag}(v))$
```

Proof. We conduct the proof in multiple stages. First, we show that the battery's capacity constraints are complied with. In the second part, we give proof that all labels of the target node's bag are part of the solution, and no element of the solution is not part of the algorithm's result. We base our proof on the correctness of multi-criterion search, introduced in Algorithm 2.2.

Part 1: Battery Constraints. Lines 15 to 17 of Algorithm 3.2 ensure that no label is merged that falls outside the battery capacity constraints by immediately discarding labels that deplete its charge, and limiting recuperation to prevent overcharging. The start vertices initial label fulfils these conditions, and all other labels to be merged are subject to said check. Thus, the battery capacity constraints are intact.

Part 2: The labels in each node's bag form its Pareto front. Due to the correctness of Algorithm 2.2, we only need to show that our extensions do not violate it. We have already given proof that our restrictions on battery capacity range are accurate, so it remains to show that our method of iterating over consumption tables produces correct results. This is easy to see, as for each edge scanned, *all* possible speeds that are allowed on this edge are examined, and merged if they yield an improvement. No label is discarded erroneously, and each label merged is taken into account in the queue, as we update the bag's key in the queue at the end of an edge scan if a label has been merged successfully. Thus, as all labels are examined in the right order—this follows from the correctness of Algorithm 2.2—and each label is created and merged correctly, all non-dominated labels created at each vertex form its Pareto front. Because the bag data structure automatically discards dominated labels, the set of non-dominated labels is exactly the set of Pareto-optima. □

4. Running Time Optimisations

In the previous chapter, we introduced our basic algorithm and hinted at several points where potential for tuning exists. In fact, the number of labels created in the naïve implementation is large enough to yield unacceptable running times, even on small road networks and with low battery capacity. To reduce practical running time to a manageable extent, we now present several techniques to speed up computation in various non-heuristic, i.e. result-preserving, and heuristic ways, i.e. ones that trade in some result quality for speed-ups. Section 4.1 covers improvements upon the basic algorithm that are known from literature, while Section 4.2 introduces two novel approaches that are applicable in our case. Section 4.3 introduces techniques for discarding labels on-the-fly, yielding further running time improvements. We also present some details on our implementation in Section 4.4.

4.1 Existing Speed-Up Techniques

We begin by transferring already known techniques for speeding up the generic multi-criterion search algorithm from literature to our approach. A number of non-heuristic running time optimisations that maintain correctness exist to improve the practical running time of multi-criterion search [DMS08, DW09], three of which we introduce in the following.

4.1.1 Hopping Reduction

We observe that it is never optimal to return to the previous vertex, i.e. “turn around”, as long as negative cycles do not occur in the graph, which we have established previously. Thus, we can abort an edge scan if we detect this behaviour: before iterating over the edge’s consumption table, we check whether the current label’s parent label is stored in the head vertices bag and abort if this is the case [DMS08]. This is called *Hopping Reduction.*

4.1.2 A* Search

Similarly to the single-criterion case, we can direct the search towards the target with A* search. To do so, we need to use a multi-dimensional potential function, providing lower bounds for each criterion [SWI91]. In our scenario, this function is two-dimensional and yields lower bounds on travel time and energy consumption. We then add its values to the bags’ keys.

Here, we obtain a very good potential function by performing reverse Dijkstra searches from the target vertex. In the first search, we determine the set of vertices that can reach the target with the given battery capacity and the minimum energy consumption required to do so. The second search operates on the vertices visited by this first run and computes minimum travel times to each of them. The combination of these values is then used as the nodes' potential function. As the values are obtained by a label-correcting reverse search, they automatically provide a feasible potential function—negative reduced weights would imply that the vertex could be reached more cheaply via the edge, contradicting the correctness of Dijkstra's algorithm. Vertices that were not visited in the first search are assigned an infinitely high potential as the battery capacity is insufficient to reach them. In the queue, we add these potentials to the bags' keys, so that a bag with key, i.e. most energy-saving label, (t, c) and potential (π_t, π_c) is assigned key $(t + \pi_t, c + \pi_c)$ in the queue. Using these node potentials, the search is drawn to the target very quickly, leading to the discovery of non-dominated paths to the target very early in the execution of the algorithm. As it results in edge weights with non-negative values in each criterion, the the number of labels created during execution is reduced dramatically. If previously, a potential shifting technique as described in Section 2.2.1 for the uni-criterion case was used, this is no longer necessary. This remedies our previous concerns about running time with recuperation in Section 3.3.

4.1.3 Target Pruning

On a graph with non-negative edge weights, if we have already found a path to the target, i.e. the target node's bag is not empty, we can use this bag to *prune* the search. Clearly, it is not viable to expand the edges out of a node if the current label is already dominated by the target node's bag. We can implement this by adding a check for this behaviour to the algorithm, just before beginning to scan the outgoing edges, called *Target Pruning*. This technique is sometimes also referred to as *Dominance by Early Results* [DMS08]. We observe that Target Pruning is not applicable if recuperation occurs in the graph, as the resulting negative energy consumption may reverse the domination by the target bag, even if A* Search is enabled, as it only modifies the bags' *keys* and not the edge weights. We can, however, leverage the A* potential function to overcome this problem.

Extended Target Pruning As the A* potential function described above provides lower bounds for each criterion, we can use it to solve the issue with Target Pruning and recuperation. To achieve this, we add the current node's potential to the label in question. If this virtual label is dominated by the target vertices bag, i.e. could not be merged into it, we can safely prune the search tree. Not only does this allow the use of Target Pruning in graphs with negative values in some or all of the edge weights' criteria, it also provides a stricter criterion for the general case, as the search is pruned not only if the label is dominated by the target, but also if the best achievable result that could arise from it would be dominated.

Theorem 4.1. *Extended Target Pruning does not discard any label that may be parent to a non-dominated label in the destination node's bag.*

Proof. Our potential function gives the lowest possible energy consumption and travel time from any reachable vertex to the destination, as they originate from reverse Dijkstra searches using minimum values as weights for each edge. Therefore, adding a node's potential to a label of the same node gives a lower bound on both energy consumption and travel time from the source to the target via the label's prior path. If this hypothetical label is dominated by the target node's bag, the label in question cannot yield a non-dominated label in the target node's bag, as its energy consumption or travel time would have to be lower than that of the dominated hypothetical label for it to be merged. As this is impossible, we can safely prune the search tree and discard the label. □

4.2 Further Improvement

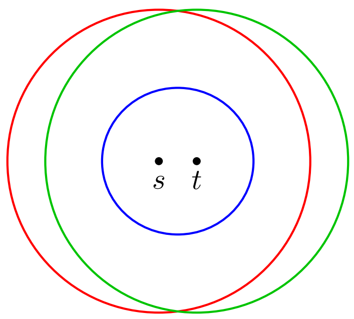
In the following, we introduce two new approaches to speeding up computation that are at least partly specific to our modelling. The first of these is Subgraph Extraction, which removes irrelevant parts of the graph to improve locality, while the second one is a heuristic technique to reduce the number of labels merged.

4.2.1 Subgraph Extraction

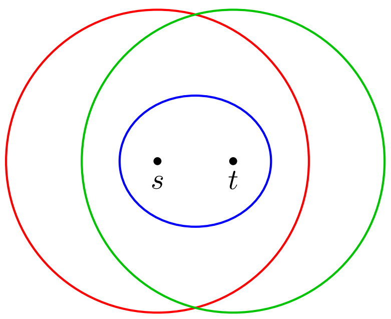
Without further additional pre-processing, the graph's vertices and edges are not ordered in any particular way. Since our queries are computationally expensive, it can be of advantage to extract the subgraph of nodes that are relevant to our query in advance. We achieve this by running a minimum-consumption A* search from the source vertex with the potential function π_c obtained during calculation of the A* potential for the main algorithm in Section 4.1.2. We restrict the forward search to the nodes visited by this reverse search, and the minimum consumption values obtained shall be denoted $\bar{\pi} : E \to \mathbb{R}$. We then extract the subgraph implied by the relevant nodes, storing them in the order in which the search settled them—recall that due to the properties of a feasible A* potential function, edge weights are non-negative and the search thus is label-setting. A node v is deemed *relevant* if the total minimum energy consumption from the source to the target via the node v, i.e. $\bar{\pi}(v) + \pi_c(v)$, is less than the battery capacity or the energy required on the *fastest* path from the source to the destination—as no path can be faster, no Pareto-optimal path can have higher consumption—, whichever is smaller. A schematic representation of search spaces is given in Figure 4.1. This procedure results in a small, cache-efficient graph, decreasing overall running time at the cost of some overhead for the extraction procedure, with the effect being most noticeable on the most computationally expensive of queries. Their search space is limited to a small corridor between source and target, see Figure 4.1(c). Observe that with the reverse search performed for subgraph extraction and A* search, we can detect unfulfillable queries, i.e. ones for which the battery's capacity is insufficient, and abort execution before the start of the main algorithm, thus avoiding searching all reachable nodes in vain.

Summary of Searches Performed During Initialisation In total, we run three searches on the graph in the initialisation phase of a query: Two reverse Dijkstra searches from the target node and one forward A* search from the source vertex. First, a reverse Dijkstra search is started at the target vertex, operating on minimum energy consumption values. Due to negative edge weights, this search is label-correcting. We halt it once no more vertices can be reached without exceeding the maximum energy consumption. As battery capacity is strictly limited, this procedure is not computationally expensive. This search provides one half of the A* potential for the main algorithm and is also used for Extended Target Pruning as well as subgraph extraction. Next, on the vertices visited by this first search, we run another reverse Dijkstra search, again from the query's destination, this time on minimum travel times. We abort this search once all vertices settled by the first search have been removed from the queue. This provides the second half of the A* potential function, which is also used for Extended Target Pruning. Lastly, a *forward* A* search is run on the same set of vertices, using minimum energy consumption as weights, starting from the source node. Its potential function is provided by the minimum energy consumption values to the target computed in the first reverse search. We abort it once no further vertex can be reached with the given battery capacity. The results of this search, combined with those of the first one, are used for subgraph extraction.

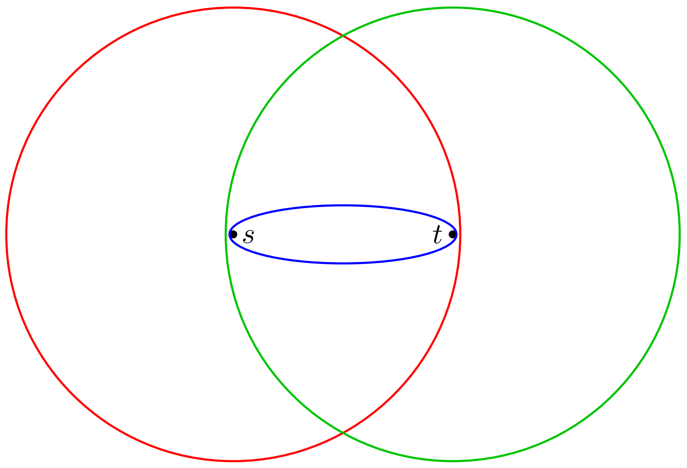
The optimisations presented up to this point are all deterministic and do not alter the result in any way. To speed up the search even more, we now turn to a heuristic technique.



(a) Short range query



(b) Medium range query



(c) Long range query. Note the extremely limited portion of the two search spaces being extracted.

Figure 4.1: Schematic representation of subgraph extraction using battery capacity as limiting factor. Extracted subgraph drawn in blue, areas reachable from s and t in red and green, respectively.

4.2.2 Early Aborting

In Early Aborting, an edge scan is aborted as soon as a label for one speed level could not be merged into the head vertices bag. The idea behind this decision is that if one label is dominated, it is likely that the same holds for the others. In the best case, the edge is absolutely irrelevant and could not yield an non-dominated label. With Early Aborting, only the first label is tested, fails to merge, and the edge scan is aborted, saving several other failed attempts at merging labels. In less apt situations, the scan may be aborted, although a subsequent speed level might provide an improvement. Clearly, Early Aborting is a heuristic technique, but our experiments showed that result quality is affected negligibly (refer to the Experiments Chapter for detailed results), while the impact on performance was much higher, as many label creations are saved. The impact on result quality is as low, as a label causing abortion of an edge scan has to be very similar to already existing ones to prevent creation of non-dominated labels. As edges are typically rather short in road networks, the impact on the Pareto front is quite small. We observe that both the speedup and the quality of the result depend on the order in which speed levels from the consumption table are processed. In our initial experiments, processing consumption table entries in descending order of driving speed yielded the best results, although more experiments are needed to confirm this.

A pseudocode implementation of our algorithm with all of the above tuning techniques included is listed in Algorithm 4.1.

4.3 Label-Discarding Heuristics

We now present two methods of heuristically discarding labels during execution of the algorithm to greatly improve efficiency while maintaining good result quality. The rationale behind the decision to discard labels is that their number is high, but at the label density of our algorithm, the informational value per label is low. As for each combination of a label and an edge that is scanned, typically two to five new ones are created, some of which are discarded immediately, but many are not, enormous numbers of labels are merged, often with only minor differences. This results in very long execution times, of which a great part is spent on minor details due to the propagation of extremely similar labels. Therefore, we have devised strategies to discard as many labels not yielding noticeable improvements as possible, while trying to minimise the number of relevant labels erroneously removed, i.e. avoiding loss of result quality. These methods operate on a single bag's set of labels and are run periodically during execution, typically once every 2^{10} to 2^{16} vertex scans. We restrict discarding to bags that were modified since the last discarding iteration to avoid loss of result quality through iterated application of the procedure. Our first approach is to discard labels based on simple pairwise similarity comparison between labels of the same parent node. The second option uses a data clustering method known from literature to determine groups of very similar labels and discard most of them.

4.3.1 Similarity-Based Label Discarding

In this technique, we compare neighbouring labels that are similar to each other, judging which of the two is more valuable. When comparing a label to its predecessor, its *value* is the additional energy consumed divided by the savings in travel time—remember that bags are ordered by ascending energy consumption. If a label's value is *low*, that means that it provides a high speed-up at moderate energy cost, making it the superior choice. Otherwise, it only provides a slight speed-up at disproportionate energy cost, and is inferior to its predecessor. However, a simple comparison of neighbouring labels could discard *too much* information. Therefore, to avoid losing an entire route by discarding all labels that belong to it, we decided to limit the procedure to labels of the same parent node. We iterate over these labels and test their similarity based on energy consumption and travel time difference. If their likeness is sufficient, we discard the label deemed inferior and continue by comparing the remaining label to the next one.

Intuition This strategy is very good at detecting labels that yield very little improvement in one criterion at rather high cost in the other. But this is not even the main point—for strict similarity criteria, the impact on result quality is quite small even if we reverse the strategy deciding which label to keep. The greatest strength of similarity-based discarding is thinning out labels that are very close to each other and were spawned from the same parent vertex. This simple intuition proves sufficient to reduce the number of labels created and processed by orders of magnitude, without sacrificing more than a few per cent of result quality, and often even less.

Workings Additionally to the simple operating principle, this procedure can be implemented very efficiently: Remembering the last label for each parent vertex, a single linear scan over the bag is sufficient to find all labels to discard. In each step, we compare the current label to its predecessor, i.e. the last scanned label from the same parent node that was *not* discarded. The first label from any parent node is never discarded, as no label has been found to compare it to yet, and is instead stored as the node's last label. In the comparison procedure itself, we first need to check that the two labels meet the similarity criteria. Therefore, we compute the current label's travel time savings Δt and additional energy consumption Δc as compared to the predecessor stored. If they are within the limits, we decide which of the two labels to remove by comparing the label's *value*, defined as $\Delta c/\Delta t$, to the threshold. For lower values, the previous label is discarded and the current one saved as new previous label for its parent vertex. Otherwise, the current label is discarded and the previous value remains. Note that when discarding a label, at first we only mark it as invalid to avoid moving around data in memory during the scan. Once each label has been examined, we perform a clean-up operation on the bag, deleting all invalid labels and restoring the labels' sorted order by compacting them.

We provided a parameterised pseudocode implementation of this procedure, making use of the efficient design described above, in Algorithm 4.2.

4.3.2 Clustering-Based Label Discarding

Our second approach makes use of the *DBSCAN* [EKSX96] clustering method, which is short for *density-based spatial clustering of applications with noise*. The term *density-based* means that it clusters points based on its number of *neighbours*, i.e. points within the local search radius. As the acronym reveals, DBSCAN has a notion of *noise*—points that do not belong to any specific cluster as they are not sufficiently close to any of them. The algorithm's basic procedure is as follows: Starting from a random unvisited point, it checks if the number of neighbours is sufficient. If this is not the case, the point is classified as noise and the algorithm proceeds with the next one. In the other case, a cluster is grown from the point, including all its neighbours and for each of these neighbours, their neighbours, provided their number if sufficient, and then their neighbours, etc. This step may reclassify points previously identified as noise.

The DBSCAN algorithm operates on any kind of spatial data with a distance measure, of which it makes heavy use. In our case, however, the points are on a single strictly monotonically decreasing curve, as they form a Pareto set. Therefore, we replace the standard neighbourhood query by a linear search to the sides until the distance limit has been reached and points are no longer considered neighbours.

Parameterisation Two parameters are required for the DBSCAN algorithm, the aforementioned local search radius, and the minimum number of points that need to lie within that distance to begin growing a cluster from a point. Although Pareto fronts generally seem similar, finding a universally good combination of these parameters is non-trivial. Moreover, additional parameterisation is required to compute the *distance* between to points—as they have two criteria that cannot directly be compared to each other, a conversion factor is required to map energy consumption difference values to such of travel times. After running the DBSCAN algorithm, it remains to decide which labels of the resulting clusters to keep and which to discard. It is clear that what the algorithm considers to be noise is of great importance to us, as these are the labels that stand out. For the clusters, it has generally proven sufficient to keep every fifth in addition the first and last label of each cluster, while the rest is discarded. More advanced strategies may yield better results, but exceed the scope of this thesis.

Issues Due to the nature of the underlying DBSCAN algorithm, clustering-based discarding is weak at finding labels that provide small benefits in one criterion at disproportionately high cost in the other. To some extent, this can be thwarted with a good parameter choice, but as described above, this is a non-trivial task and it is unrealistic to expect an expert's educated guess for each query.

Additionally, the clustering step makes use of dynamic data structures in several places, e.g. for maintaining the queue of neighbours to be processed when growing a cluster. This results in running times that far exceed those of similarity-based discarding's linear scan, which uses a single small array for storing predecessors. Note that in our initial experiments, it proved counterproductive to only apply the clustering method to labels of the same origin, as that would greatly increase complexity of neighbourhood queries and further increase the number of dynamic data structure operations.

4.4 Implementation Details

In this section, we describe some details of our implementation to guide re-implementation of our methods and shed some light on how our experimental results were obtained. We begin by presenting our choices of data structures and details of their implementation, before sharing some aspects of our implementation project's structure.

Data Structures and Operations

First, we present more information on the data structures introduced in Chapter 3, and how their actual implementation differs from their description.

Priority Queue Our choice of priority queue is a binary heap tracking pointers to the vertices' bags. The bags provide an interface to compare them to each other, allowing us to implement the heap in a very generic way that does not depend in any way on its contents being bags of labels. Therefore, we can use the same priority queue implementation to organise the bags in our extension of multi-criterion search, and the road segments in single-path optimisation.

Consumption Tables As for consumption table storage, we found that many edges share the same characteristics. Therefore, we represent consumption tables in a slightly different manner than introduced in Section 3.1.1. Instead of storing travel time and energy consumption *along the edge*, consumption tables of our implementation do not store travel time, but hold the speed driven as well as the energy consumption *per metre*. Combined with the edges' lengths, we can restore the original data. This change requires us to slightly alter the procedure for merging consumption tables when contracting vertices. We can no longer add the energy consumption values, but instead need to take the weighted average according to length distribution of the edges. Also, instead of storing the consumption tables with the edges, we keep the significantly smaller set of distinct tables separately and reference them from the edges to save space in storage and execution, and leverage the power of modern processors' caches when accessing them. Our initial experiments showed that the number of distinct consumption tables is typically multiple orders of magnitude less than the number of edges, especially in large road networks.

Searches Performed in Initialisation In Section 4.2.1, we described the various searches performed on the graph during initialisation, which operate either on minimum consumption values or minimum travel times. In addition, two of the three searches are *reverse* searches, operating on the reverse graph. Therefore, we create a second graph with all edges reversed. The edge weight consists of tuples, containing the minimum energy consumption and the fastest travel time. Both reverse searches then operate on this graph, which allows for significantly higher performance in the initialisation. The graph is constructed at instantiation of the algorithm—before initialisation for the first query—, and both searches operate on the same graph to reduce running time in graph creation, and memory consumption of the graph(s). Equally, a similar graph is constructed for the forward A* search, which is required for subgraph extraction.

Project Structure

We now give an overview of the libraries used to implement our algorithms and the components of this implementation.

Libraries Heavy use is made of the C++ Standard Template Library (STL), especially `std::vector`, which is used whenever collections of data need to be stored. We also make some use of the STL's collection of algorithms such as `find` or `sort`. We have refrained from the use of inheritance in performance-sensitive places, utilising the C++ template system instead. Apart from the STL and GDAL[1], which we utilise for reading data from a Digital Elevation Model during data aggregation, no external libraries were used in our implementation.

Components Our implementation is split into several parts: Data aggregation, which we describe in Section 5.1 of the Experiments Chapter, is performed by a stand-alone utility that needs to be run only once to build the data structures necessary for execution. For running experiments, a command-line program is used that runs a number of queries and stores their results for evaluation. We also have a web-server implementation that brings our implementation to web browsers, allowing definition of the query by dragging around markers on a map and viewing the results directly in form of routes displayed in said map. Some examples of this can be seen in Chapter 6. Furthermore, a stand-alone implementation of the path optimisation procedure described in Section 3.2, applied to a set of paths and keeping only the Pareto-front of the result, can be run as a post-processing step on saved query results to study the effects of acceleration on the outcome. We have decided not to include energy and time costs and savings incurred by ac- and deceleration, respectively, in our main algorithm, as it failed to work satisfactorily due to modelling issues. For example, our model does not account for traffic lights or turns, unduly skewing the results in favour of cities, despite the typically higher energy consumption in inner-city traffic due to frequent de- and acceleration. Additional experiments with acceleration penalties did, however, turn out to have very favourable running time behaviour and should yield more realistic results with better performance when implemented with a suitable classification of roads and turns, as less labels are merged for paths deviating by a few turns only, such as taking a small street running in parallel.

[1]`http://www.gdal.org`

Algorithm 4.1: Our Algorithm with Optimisations

Input: Graph $G = (V, E, \textsc{cons}, \textsc{time})$ with $\textsc{cons}, \textsc{time} : E \to \mathbb{R} \cup \dots \cup \mathbb{R}^\ell$ as consumption tables, source node s, target node t, battery capacity maxCons
Data: Priority queue Q, bags for vertices reachable given battery constraints
Output: Pareto set and paths from the source s to the target t, with energy consumption limited to maxCons

```
   // Initialisation
 1 forall $v \in V$ do
 2     $\textsc{Bag}(v) \leftarrow \emptyset$
 3 $\textsc{Bag}(s) \leftarrow \textsc{label}(0,0)$
   // A* potential function
 4 $\pi \leftarrow \textsc{reverseSearches}(t,$ maxCons$)$
 5 if $\pi(s) >$ maxCons then
 6     RETURN('Unreachable Query')
   // Subgraph Extraction
 7 $\pi' \leftarrow \textsc{forwardConsumptionSearch}(s, \pi,$ maxCons$)$
 8 $V' = \{v \in V \mid \pi(v).\textsc{cons} + \pi'(v) \leq$ maxCons$\}$
 9 $E' = \{(u, v) \in E \mid u \in V' \land v \in V'\}$
10 Q.UPDATE($\textsc{Bag}(s), \pi(s)$)
   // Main loop
11 while Q *is not empty* do
12     $u$, bag $\leftarrow$ Q.FRONTELEMENT()
13     currentLabel $\leftarrow$ bag.POPFIRSTUNPROCESSEDLABEL()
14     if bag.HASUNPROCESSEDLABEL() then
15         Q.EXTRACTFRONT()
16     else
17         Q.UPDATE(bag, $\pi(u)$)
18     if $\textsc{Bag}(t)$.DOMINATES(currentLabel $+ \pi(u)$) then
           // Extended Target Pruning
19         CONTINUE
20     forall $(u, v) \in E'$ do
21         if currentLabel.previousLabel.vertex $= v$ then
               // Hopping Reduction
22             CONTINUE
           // Iterate over the speed levels
23         forall *levels* $n \leftarrow 1, \dots, k \leq \ell$ do
24             newHeadLabel $\leftarrow$ currentLabel $+ (\textsc{cons}(n), \textsc{time}(n))$
25             newHeadLabel.CONS $\leftarrow \textsc{max}(0,$ newHeadLabel.CONS$)$
26             if newHeadLabel.CONS $>$ maxCons then
                   // Battery charge depleted
27                 CONTINUE
28             newHeadLabel.previousLabel $\leftarrow$ currentLabel
29             if $\neg\textsc{Bag}(v)$.MERGELABEL(newHeadLabel) then
                   // Early Aborting. Heuristic!
30                 BREAK
31         if $\textsc{Bag}(v)$ *was modified* then
32             Q.UPDATE($\textsc{Bag}(v)$,$\pi(v)$)
```

Algorithm 4.2: SIMILARITY-DISCARDING

```
    Input: Label set labels with first unprocessed label index firstUnprocessedLabelIndex,
           thresholds timeThreshold, consumptionThreshold, and similarityThreshold
    Data: previousLabel to store reference to previous label of the same parent vertex
    Output: None
 1  if labels.SIZE() − firstUnprocessedLabelIndex $\leq 3$ then
        // None or too few unprocessed labels
 2      RETURN
 3  previousLabel.PUSH(labels(firstUnprocessedLabelIndex))
 4  for firstUnprocessedLabelIndex $< i <$ labels.SIZE() do
 5      index $\leftarrow 0$
 6      while index $<$ previousLabel.SIZE() $- 1$ and
        previousLabel(index).PREVVERTEX() $\neq$ labels($i$).PREVVERTEX() do
 7          index $\leftarrow$ index $+ 1$
 8      if previousLabel(index).PREVVERTEX() $=$ labels($i$).PREVVERTEX() then
            // Compare labels
 9          $\Delta t \leftarrow$ previousLabel(index).ARRIVALTIME() $-$ labels($i$).ARRIVALTIME()
10          $\Delta c \leftarrow$ labels($i$).CONSUMPTION() $-$ previousLabel(index).CONSUMPTION()
11          if $\Delta t <$ timeThreshold or $\Delta c <$ consumptionThreshold then
                // Labels are within discarding range
12              if $\Delta c / \Delta t >$ similarityThreshold then
                    // High additional consumption per time saved
13                  labels($i$).SETVALID(false)
14              else
                    // Little additional energy consumption per time saved
15                  previousLabel(index).SETVALID(false)
16                  previousLabel(index) $\leftarrow$ labels($i$)
17          else
18              previousLabel(index) $\leftarrow$ labels($i$)
19      else
            // This label's parent vertex has not occurred before
20          previousLabel.PUSH(label)
21  DELETEINVALIDLABELS()
```

5. Experiments

In this chapter, we present experimental results of our implementation. First, we describe the environment, input data, and experiment methodology in Section 5.1, before evaluating the performance of the baseline algorithm and our running time improvements in Section 5.2.

5.1 Setup

We implemented all algorithms in C++ using clang++ 3.2 with optimisation level 3 as compiler. Experiments were conducted on a machine with dual Intel Xeon E5-2670 CPUs clocked at 2.6 to 3.3 GHz, with 20 MiB of shared L3 cache, 256 KiB of L2 cache per core, and 32 GiB of DDR3-1600 main memory for each node. To achieve reproducible running times, only one instance was running at any given time.

5.1.1 Input Data

When describing our modelling in Section 3.1, we did not provide any specifics on the input data required to create the network. Here, we describe the metadata needed and the data sets used to obtain this information for our experiments. First, we require the road network to provide geographic coordinates for vertices, and information about edges' length, type, and speed limits. As we use discrete speed values and pre-compute energy consumption, a typical choice of speeds could correlate to speed limit road signs, as these provide intervals that are neither too coarse nor to specific to be realistically satisfiable. Furthermore, we define a minimum speed for each road type that needs to be abided by to avoid becoming an obstacle to other road users. Including the aforementioned speed limit information, we now have the set of speed values for edges' consumption table. As energy consumption greatly depends on road gradients, we need to integrate slope information. Hence, we enrich this data with height information from a *Digital Elevation Model* (DEM) and compute slopes for the roads, using the coordinates from the road network. In the following, we call the combination of road type, possible speed levels, and slope the road segment's *road class*. We combine this data with a consumption model to compute the consumption tables. We have visualised some examples of these in Figure 5.1 and describe them on the next page.

Conditions for Compression of Vertices

As we now have the necessary metadata, we can decide on the conditions for compressing a vertex. Recall that in Section 3.1.1, we introduced the idea of disallowing speed changes if the road conditions remain the same. Here, we can specify these requirements. First of all, a vertex can only be deemed compressible if it has exactly two neighbours, i.e. represents a stretch of road, as intersections represent a change in road conditions. Moreover, both road segments need to be of the same category and have the same possible speed levels. This ensures that we do not compress nodes where the road changes and compression would create faulty results, as we cannot assume conditions to be the same. Furthermore, to avoid incorrect results due to overcharging with recuperation or running dry of energy uphill, tables cannot be merged if the energy consumption values' signs differ [SLAH11], i.e. one is recuperating energy and the other is not.

Experimental Data

For experimental evaluation, we used a road network kindly provided by PTV AG for scientific use to obtain information about road segments, road types, intersections, and geographical coordinates. To obtain information about slopes, we integrated the Shuttle Radar Topography Mission (SRTM) data in version 4.1 provided by the CGIAR Consortium for Spatial Information[1], which is based on NASA's SRTM3 Digital Elevation Model (DEM). We chose the post-processed data from the CGIAR Consortium over NASA's as any missing points in the original data have been added from alternative sources or interpolated with advanced algorithms, as well as due to better overall consistency. It covers large parts of the world with a resolution of three arc-seconds, which corresponds to a distance of 90m at the equator. We deleted all areas for which no height information was available from the graph, removing large parts of Norway, Sweden, and Finland, as their area stretches beyond the 60th circle of latitude, where the data ends. We also removed private roads and ferries and extracted the largest strongly connected component from the remaining graph, consisting of 19,046,204 nodes and 44,675,948 edges after contraction of nodes with two neighbours, as detailed in Section 3.1.1.

The energy consumption data for our experiments originates from PHEM (Passenger Car and Heavy Duty Emission Model) [HRZL09], developed by the Graz University of Technology. Among others, PHEM contains electric vehicle energy consumption values for a large variety of driving situations, including road categories, speed limits, slopes, and traffic situations. A heuristic to map this data to the input road network was kindly provided by the authors of [BDP]. Their technique measures the similarity between road segments of the PTV data and the parameters of PHEM. As no information about traffic situations was available to us, we assumed unobstructed traffic on all roads when assigning consumption data. The vehicle chosen for our experiments was a Peugeot iOn, a battery-electric city car manufactured by Mitsubishi Motors, for which highly detailed consumption data is available in PHEM. Nominally, it has a battery capacity of 16 kWh, which is electronically limited to 70–80 % of the full range to improve durability. We did not include these limitation of the production vehicle in our experiments, allowing arbitrary values to be specified for the vehicle's usable battery capacity.

Sample Consumption Tables

We have plotted a number of exemplary consumption tables as a heat map in Figure 5.1, which were computed with the mapping procedure described above. Each speed value represents a table of its own, so that a total of 33 consumption tables are shown. For example, the four dots in the rows labelled "Rural" and the column "-4 %" might form a consumption table for a rural road with a slope of -4 %, providing energy consumption values for speeds from the hypothetical speed limit of 80 km/h down to 50 km/h.

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

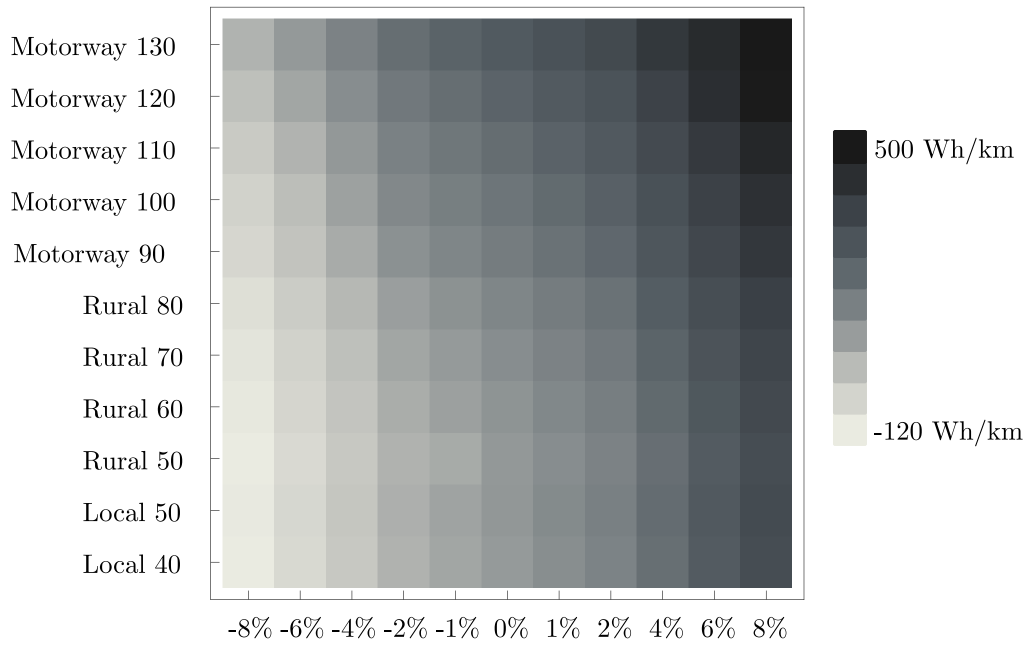


Figure 5.1: Sample energy consumption values by road class and gradient, as heat map. Speed values are given in km/h.

A number of observations can be made based on these tables. Firstly, we see that driving speed has a significant impact on energy consumption. Furthermore, our vehicle, a Peugeot iOn, recuperates energy more efficiently at lower speeds. Also, less energy is required at the same speed on rural roads than on local roads—the row is in slightly lighter shades of grey. We attribute this to the typically smoother driving style on non-urban roads, which requires less acceleration. Lastly, we clearly see that the effects of avoiding steep climbs are very noticeable.

Choice of Parameters

Some of our methods, namely the label discarding procedures described in Section 4.3, require additional parameters. The values used here were determined in initial experiments and we do not make any claim of optimality, as they simply represent choices that worked well for our data.

For similarity discarding, where labels that differ only by a certain amount in one or both criteria are compared to each other, three parameters are required. Apart from the maximum difference in energy consumption and travel time, respectively, we need to specify the threshold by which to judge which label is better. While similarity discarding is great at detecting labels that yield minor improvements in one criterion at unreasonably high cost in the other, we also need to take care not to discard too much information. We therefore chose a time difference of one second and energy consumption of three Watt-hours as similarity criteria, and a threshold of 2.5 Wh/s (9kW) to determine which label to keep. While these values seem overcautiously low, initial experiments showed them to yield better results than higher values, which discarded too many labels and resulted in a loss of precision.

In clustering-based discarding, choosing the parameters for the DBSCAN algorithm constitutes the biggest challenge. As stated in Section 4.3.2, a universally satisfying choice is hard to find, as good parameters are at least in part specific to each individual query or even bag. Our initial experiments showed that requiring two points within a neighbourhood of 1000 units works for most queries, where a *unit* is either one mWh in energy consumption or 1/2500th of a second. The distance of two labels $l_1 = (t_1, c_1)$ and $l_2 = (t_2, c_2)$ is thus defined as $d(l_1, l_2) = \sqrt{|250(t_1 - t_2)|^2 + |c_1 - c_2|^2}$, assuming that travel times are given in tenths of seconds and energy consumption in milliwatt-hours, as is the case in our implementation. Again, higher values for the thresholds lead to an increased loss of result quality.

5.1.2 Query Generation

Due to the restricted range of electric vehicles, the common method of picking a random source and destination vertex is unsuitable in our case. Instead, we start a search from a random vertex, using minimum energy consumption on the edges as their weight, and abort once the battery is depleted. We then randomly select a target vertex from the search space and restart the procedure to generate the next query. After computing the desired number of queries, we save them and run all instances on the same set of queries to obtain comparable results.

5.1.3 Quality Measurement

We developed a simple quality measure to evaluate our heuristic tuning techniques. When comparing a set of paths $p = \{p_1, \ldots, p_n\}$ to another set $P = \{P_1, \ldots, P_k\}$, we compute the similarity between each pair of paths from p and P. Our *similarity* measure, shown in Equation 5.1, is one minus the quotient of the length that the paths do not share and their total combined length. Therefore, each road segment that occurs in *both* paths is added twice to the denominator, while each road segment that occurs in only one path is added to both the numerator and the denominator. We call the numerator diff and the denominator len. For each path p_i of p, we pick the corresponding path $P_{\text{match}(p_i,P)}$ of P that has the highest similarity, see Equation 5.2. We then sum up all the diff values, divide them by the sum of the len values, and subtract the result from one to obtain the relative quality qual(p, P) of the path p compared to P, as shown in Equation 5.3.

$$\begin{aligned}
\text{diff}(p_i, P_j) &= l(p_i \setminus P_j) + l(P_j \setminus p_i) \\
\text{len}(p_i, P_j) &= l(p_i) + l(P_j) \\
\text{sim}(p_i, P_j) &= 1 - \frac{\text{diff}(p_i, P_j)}{\text{len}(p_i, P_j)} && (5.1) \\
\text{match}(p_i, P) &= j : \text{sim}(p_i, P_j) = \max_{1 \le j \le k} \text{sim}(p_i, P_j) && (5.2) \\
\text{qual}(p, P) &= 1 - \frac{\sum_{1 \le i \le n} \text{diff}(p_i, P_{\text{match}(p_i,P)})}{\sum_{1 \le i \le n} \text{len}(p_i, P_{\text{match}(p_i,P)})} && (5.3)
\end{aligned}$$

5.2 Evaluation of Running Time Optimisations

In this section, we evaluate the speed-up techniques introduced in Chapter 4. First, Section 5.2.1 assesses the effects of the general tuning methods, which we presented earlier in Section 4.1. Following this, we analyse the running time effects and impact on result quality of our novel label discarding heuristics introduced in Section 4.3. This second part is presented in Section 5.2.2.

5.2.1 Running Time Effects of Basic Tuning Methods

To allow for reasonable evaluation of the unoptimised instances, we chose to set the vehicle's maximum consumption to 4 kWh for this first round of experiments. We then generated

100 random queries with the method detailed above. Table 5.1 shows the resulting running times with the initialisation overhead subtracted. On large road graphs, initialisation of the algorithm accounts for a significant portion of the running time for the optimised versions of the algorithm due to the low range in these queries. In our experiment setup described above, we measured an average 230 ms without A* potential calculation, which required around 5 ms extra on average. The initialisation times include subgraph extraction, which requires consumption potential calculation even if A* search is disabled. Note that A* search is considered a requirement for Extended Target Pruning, as it makes use of both consumption and travel time potentials, the latter of which would not be computed otherwise.

Methodology The results of our experiments evaluating the different tuning parameters are listed in Table 5.1. The same set of 100 random queries on the road network of Western Europe were conducted in each case, with a battery capacity of 4 kWh, for which a value this small was chosen in order to provide acceptable running times even for the naïve implementation without any tuning. The left side of the table details which running time optimisations were enabled per run, while the right side shows the average running time and number of comparisons performed as well as the speed-up provided over the naïve version. Comparison operations are counted in the priority queue as well as in label domination in the bags. Both are incremented for each comparison performed between two elements of the data structure.

Results Evaluating the results, we clearly see the value provided by A* search through its implicit potential shifting. As the algorithm does not spawn labels that dominate already processed ones if all edge weights are non-negative in each component, much fewer labels are created and dominated soon after. We measured a nearly thirty-fold decrease in running time while 46 times less comparisons were performed. Also interesting is the rather high speed-up that the seemingly primitive technique of Hopping Reduction (HR) yields: Compared to the unoptimised version, running times are shortened by a third, and even with all other non-heuristic tuning flags enabled, it still yields nearly 15 per cent improvement in running time and even more in the number of comparisons performed.

The effect of Extended Target Pruning is also very prominent: Running times dropped by an additional factor of more than 20 when enabling ETP compared to A* search alone, while the number of comparisons decreased to around one fifteenth of its previous value. When additionally enabling the heuristic Early Aborting (EA) technique, we saw yet another speedup by a factor of 2.7, while maintaining near perfect result quality. Note that the other techniques evaluated in this series of experiments are all non-heuristic and thus do not yield any changes to the result.

Table 5.1: Running times without label discarding at a battery capacity of 4 kWh. The same 100 random queries were run in all instances. HR is Hopping Reduction, A* goal-directed search, ETP Extended Target Pruning, and EA Early Aborting.

Tuning Flags				**Query**			
HR	A*	ETP	EA	[ms]	Speedup	Avg comparisons	Quality
×	×	×	×	632 416	1.0	465 433 555 653	1.0000
✓	×	×	×	421 252	1.5	300 115 332 047	1.0000
×	✓	×	×	22 516	28.1	10 112 644 561	1.0000
×	✓	✓	×	995	635.6	692 908 967	1.0000
✓	✓	✓	×	861	734.5	568 855 705	1.0000
✓	✓	✓	✓	324	1 951.9	212 071 184	0.9993

Table 5.2: Comparison of result quality for label discarding methods. The column # specifies the frequency of label discarding as logarithm to base 2; EA refers to Early Aborting; the values in the *Quality* column are result quality compared to the non-heuristic version, while Q_{EA} is quality relative to the version with EA.

Discarding		**Without EA**		**With EA**		
Mode	#	Quality	Time [s]	Q_{EA}	Quality	Time [s]
None	—	1.0	777.23	1.0	0.997	267.12
Similarity	18	0.997	50.57	0.996	0.995	24.45
	16	0.993	23.04	0.992	0.992	11.57
	14	0.987	11.36	0.986	0.987	6.02
	12	0.981	6.93	0.980	0.982	4.10
	10	0.973	5.13	0.974	0.976	3.49
	8	0.970	4.87	0.968	0.971	3.52
Clustering	14	0.974	84.20	0.976	0.977	36.15
	12	0.962	50.90	0.963	0.966	20.36
	10	0.955	35.65	0.952	0.955	16.27
	8	0.944	31.24	0.942	0.947	19.58

Target Pruning Issues As our model includes recuperation and thus negative edge weights, regular Target Pruning is not applicable, as described in Section 4.1.3. We did not list experiments that evaluate its effects on running time, as ignoring negative energy consumption values to enforce non-negative edge weights would have produced results that would not be meaningful in comparison to other results, nor would they be practically relevant. Still, experiments with the same input data as was used throughout this chapter yielded a speedup factor of 10 for Extended Target Pruning over regular Target Pruning, both with Hopping Reduction and A* search enabled. We note that while the results computed in this experiment are not technically correct, these measurements allow us to roughly quantify the relevance of Extended Target Pruning.

5.2.2 Evaluation of Label Discarding Procedures

In Section 4.3, we introduced two label discarding procedures, which we will now refer to as *Similarity* and *Clustering*. The results of experiments conducted with these techniques at different discarding frequencies—shown as logarithms to base two in the column labelled #—in combination with and without Early Aborting (EA), are listed in Table 5.2. The discarding frequency specifies the number of vertex scans that pass between two iterations of label discarding. For each discarding technique and frequency, the table provides average running time and result quality for the same set of 1000 random queries in the road network of Western Europe at a battery capacity of 16 kWh, generated with the method detailed above. In the column for experiments with Early Aborting enabled, we provide comparisons with the non-heuristic algorithm not using any label discarding technique, with and without Early Aborting enabled, referred to by subscripts *EA* and *base*, respectively. Additionally, Table 5.3 shows the development of running times and number of comparisons performed for the same set of queries. For each clustering mode, we give average running times and number of comparisons (in billions), as well as the factors by which they these numbers are smaller than those for their non-discarding counterparts. Again, we compare the Early Aborting versions with both versions of the non-discarding algorithm, with and without Early Aborting enabled.

Comparison of Discarding Techniques Shown in Figure 5.2 is a visualisation of Table 5.2, showing result qualities compared to the base algorithm. It clearly shows that

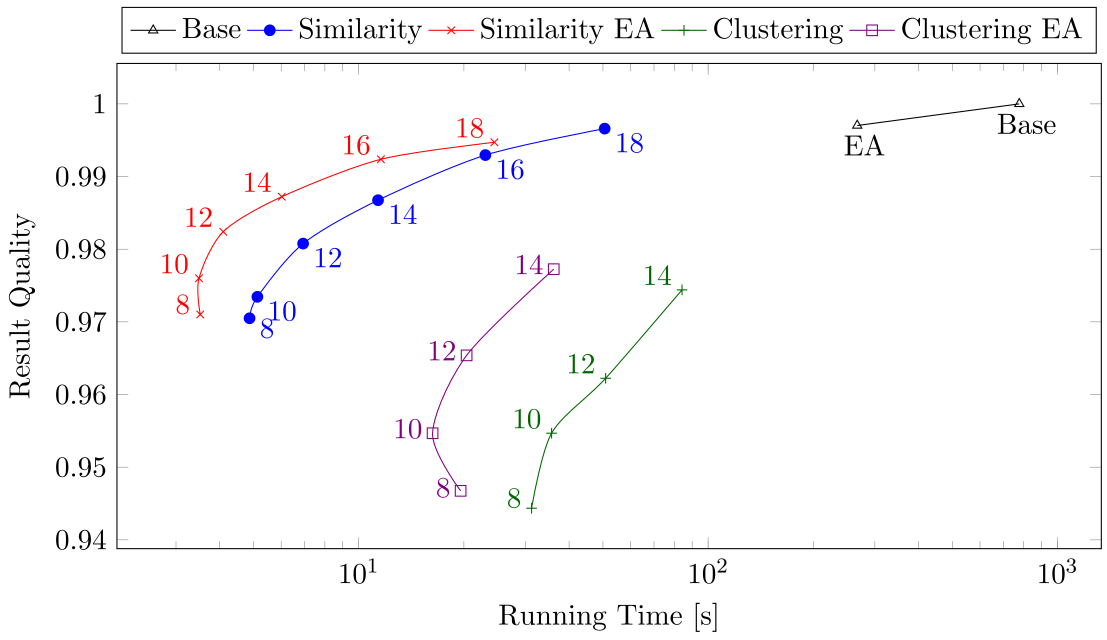


Figure 5.2: Running times and result qualities of discarding modes, as shown in Table 5.2.

Similarity is superior to *Clustering* in both running time and result quality. For running time, the explanation is quite simple: *Clustering* discards a lot less labels. For example, at a discarding frequency of 2^{12}, which would be a good choice both speed- and quality-wise for either method, the average number of comparisons performed by *Clustering* is 6.2 times higher than that of *Similarity* without Early Aborting, and 4.4 times higher with it enabled. This explains most of the difference in running time, the other part of which stems from the higher complexity of *Clustering*, which has to perform a much higher number of operations, some of which involve dynamic data structures, requiring memory allocations and copying of data. The $\Delta C/s$ column of Table 5.3 shows this well. When applying *Clustering* frequently, the algorithm performs less queue or bag operations, i.e. comparisons, per second than the base algorithm, as much of the time is spent on discarding labels. On the other hand, *Similarity* actually increases throughput, as less time is required to maintain consistency in smaller bags. We can also clearly observe that for low discarding frequencies, the running time of the discarding procedures factors visibly into the total execution time. This is particularly prominent in the Early Aborting versions, as these merge less labels and therefore "do less work" in between discarding runs. Especially for *Clustering*, it can be preferable to run the procedure *less often*, as the overhead of executing DBSCAN more frequently exceeds the benefits of discarding more labels, and the less frequently discarding algorithm computes more accurate results. See *Clustering* with Early Aborting at discarding frequencies 2^8 and 2^{10} in Figure 5.2.

Result Quality When regarding Figure 5.2, we clearly see that the result quality of *Similarity* is excellent. For discarding frequencies above 2^{10}, the average result quality was consistently above 98 %, and the median exceeded 99 %. Less than 2 % of queries had a result quality below 90 %, and under 9 % fell below 95 % percent of quality, while a full fifth of queries scored a result quality of over 99.9 %. A quantile plot, comparing these results against a uniform distribution, is shown in Figure 5.3.

Combining Label Discarding with Early Aborting When comparing the results of the *Early Aborting* and standard versions of a discarding mode, we made an interesting observation. Their quality values are remarkably similar, with the *Early Aborting* version

Table 5.3: Running times and number of comparisons performed at various discarding settings in the same experiments as in Table 5.2. Column # contains discarding frequency as base-2-logarithm. Comparison counts in billions are given in column # C. Column S contain the speedup, while column $\Delta C/S$ lists by how many percent the throughput in comparisons per second increased over that of the base algorithm. Speedup is compared to both Non-EA and EA versions in columns S_{base} and S_{EA}, respectively.

Discarding		**Without EA**				**With EA**				
Mode	#	[s]	# C	S	$\Delta C/S$	[s]	# C	S_{EA}	S_{base}	$\Delta C/S$
None	—	777.23	494.3	1.0	0.0	267.12	170.9	1.0	2.91	0.6
Similarity	18	50.57	38.52	15.4	19.8	24.45	17.56	10.9	31.8	12.9
	16	23.04	18.19	33.7	24.1	11.57	8.64	23.1	67.2	17.4
	14	11.36	9.12	68.4	26.2	6.02	4.53	44.3	129.0	18.3
	12	6.93	5.30	112.1	20.2	4.10	2.91	65.2	189.8	11.7
	10	5.13	3.83	151.5	17.4	3.49	2.39	76.5	222.7	7.8
	8	4.87	3.34	159.6	7.6	3.52	2.20	75.9	220.8	-1.5
Clustering	14	84.20	60.46	9.2	12.9	36.15	24.20	7.4	21.5	5.2
	12	50.90	32.77	15.3	1.2	20.36	12.87	13.1	38.2	-0.6
	10	35.65	19.79	21.8	-12.7	16.27	8.66	16.4	47.8	-16.3
	8	31.24	15.47	24.9	-22.2	19.58	9.13	13.6	39.7	-26.7

sometimes even being ahead of its regular counterpart. We explain this behaviour with the similarity in the intuition of Early Aborting and label discarding—both techniques aim to reduce the number of labels that have negligible impact on the result. The difference is solely in when they are applied. While Early Aborting prevents the inclusion of some such labels before they can delay the algorithm's progression, label discarding techniques are run intermittently to clean up those labels that made it past this first barrier. Therefore, Early Aborting should always be enabled when a label discarding technique is used. Figure 5.2 shows that this recommendation results in significantly lower running times without any noticeable impact on result quality.

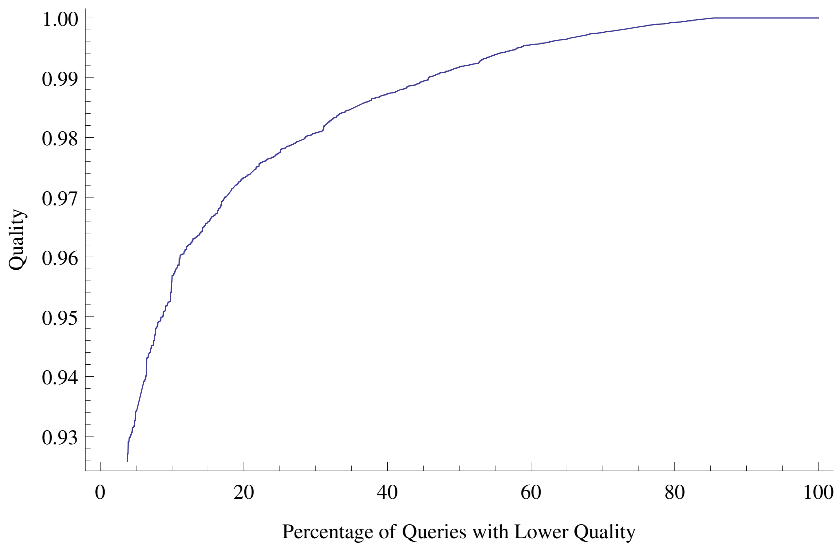


Figure 5.3: Quantile Plot for *Similarity*, performed every 2^{12} iterations.

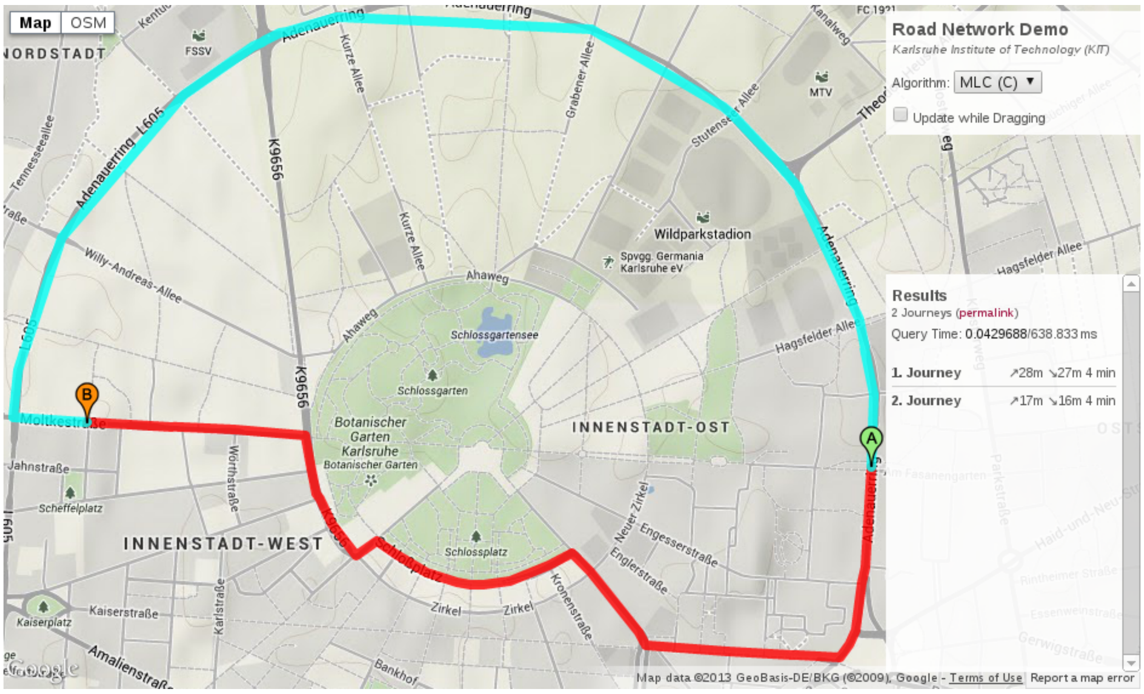
6. Case Study

In this chapter, we present some exemplary cases to illustrate the routes computed by our algorithm. These experiments should not be considered as running time measurements, which are provided in Chapter 5. Instead, they serve to convey an idea of the nature of our algorithm's results. As such, they were conducted on a Lenovo ThinkPad T420 laptop featuring a dual-core Intel Core i5-2520M processor with hyper-threading, clocked at 2.5–3.2 GHz and accessing 256 KiB of L2 cache per core as well as a shared L3 cache of 3MiB. The machine is equipped with 8GiB of DDR3-1333 main memory. Unlike the previous experiments using the road network of Western Europe, due to memory constraints, we had to limit the input graph to the road network of Germany. The other input data, i.e. PHEM and the Digital Elevation Model, was the same, however, and the resulting graph consisted of 4 464 850 vertices and 10 191 352 edges. Unless otherwise noted, queries were executed with a battery capacity of 16 kWh, with Hopping Reduction, A* Search, Extended Target Pruning and Early Aborting enabled and Similarity Discarding performed on the bags every 2^{12}th vertex scan. The queries' results are visualised as an overlay on top of a Google Maps layer to give reference to map data and terrain. As we did not find a suitable way to show driving speed in a map, our visualisation is limited to showing routes, and information on speeds driven is given textually where relevant or in the form of a Pareto front plot. Remember that running times in this setup only serve to give a rough estimate of the order of complexity. For replicability, we give the latitude and longitude of the origin and the destination in decimal degrees for each query.

Key In the screenshots below, execution times are given in two figures, first the query running time, and then the duration of initialisation, which depends mostly on the density of the network in the query's region and the battery capacity due to the way that subgraph extraction is implemented. Routes are coloured according to their fastest label's travel time, traversing the colour spectrum from blueish red for the fastest of routes via pink, purple, blue, green, yellow, and orange to red for slow connections. The web application from which the screenshots are taken also gives estimates of ascent and descent on the routes, most of which cannot be seen here due to space limitations. We note, however, that as a general trend, faster routes feature higher ascent values than the slower, longer routes, which tend to avoid slopes, as one would expect.

Local Queries

First, we present some results for local queries in the range of few kilometres. In Figure 6.1, you can see a city query with an aerial distance of little over 2 km. The fastest route,



Road Network Demo
Karlsruhe Institute of Technology (KIT)
Algorithm: MLC (C)
Update while Dragging

Results
2 Journeys (permalink)
Query Time: 0.0429688/638.833 ms

1. Journey ↗28m ↘27m 4 min
2. Journey ↗17m ↘16m 4 min

Figure 6.1: Inner-city query in Karlsruhe, Germany, from the Informatics Building of KIT (49.0135,8.41829) to the School of Applied Sciences (49.0145,8.39051)

drawn in blue, uses a ring road to the North to quickly reach the target. However, this is a detour, allowing for a more energy-efficient and direct, but slower route, shown in red. Due to the low range of the query and the primitive travel time model used in cities—see the section on future work in Chapter 7 for more—, variation in travel time is very little. As Karlsruhe is a very flat city, energy consumption depends nearly exclusively on driving speed. Due to the extreme locality of the query, the running time of 0.04 ms is insignificant in comparison to the initialisation time of over 600 ms. As hinted at above, the amount of time spent in initialisation is dominated by edge re-weighting for the forward A* search conducted for subgraph extraction and could easily be implemented in a more efficient manner if initialisation duration were an issue.

Medium Range Queries

On medium range queries between 10 and 50 kilometres, the running time of our algorithm increases noticeably, but stays well below 100 ms for query execution. Our first example is a query from one side of Berlin to the other, shown in Figure 6.2. The fastest of routes, drawn in purple, make use of the motorway circling around the city to the West, while more direct routes, shown in green, yellow, orange and red, pass through the city centre. Some of these use large roads, such as the upper route in aquamarine, that allow crossing the city nearly as fast as via the ring road, while others, e.g. the route drawn in dark orange and light green, provide more direct connections, saving energy at the cost of travel time.

An example for a query where many alternative routes exist is from Rastatt to Karlsruhe, with an aerial distance of 23 kilometres and depicted in Figure 6.3. Among others, there is the possibility of using the motorway (pink), an A-road (blue), or passing through several small towns (green). There also exist a number of different ways into the city of Karlsruhe, as shown by the plethora of routes entering from the South. The differences in travel time are very noticeable in this query, ranging from 20 minutes via the motorway and 23 minutes on the A-road to 30 minutes and more via the small towns to the West, with the slowest route requiring 43 minutes, more than twice as long as the fastest one.

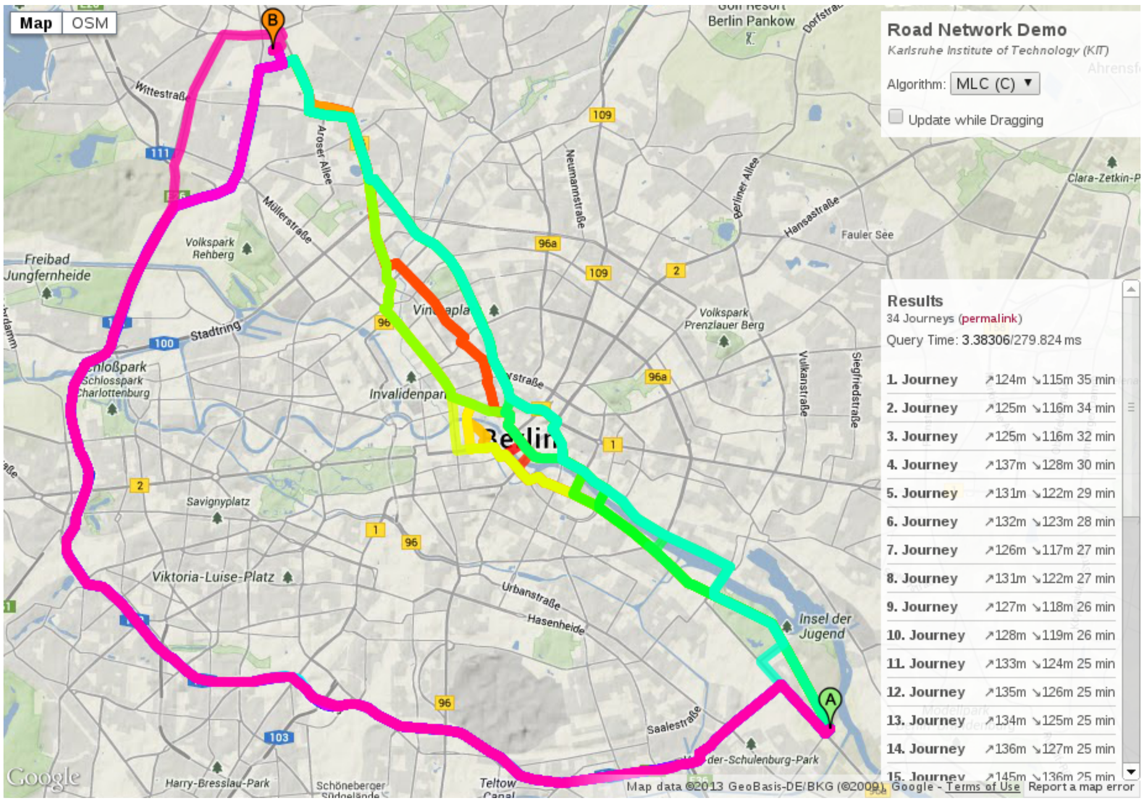
The last example for medium range queries is in hilly terrain, at the North end of the black forest, from Pforzheim to Bad Liebenzell. As shown in Figure 6.4, one can either take the road along the river Nagold—this route is shown in blue, and due to the compression of road stretches introduced in Section 3.1.1, geometric modelling is rough in some places—and reach Bad Liebenzell with rather low energy consumption. Alternative routes via elevated towns close-by, shown in purple and pink, yield faster connections at the cost of higher energy consumption.

Long queries

We now turn to queries of longer range. Here, we observe execution times of several hundred milliseconds for query execution, surpassing the initialisation time. The first example of this category is from Bad Liebenzell to Baden-Baden, shown in Figure 6.5. Again, we note the terrain, as the two cities are separated by the mountains of the northern black forest. Consequently, the fastest route, shown in pink, takes a rather big detour via Pforzheim, from there on taking the motorway via Karlsruhe, only passing through flat terrain. The pink route in the south is only two thirds of the length, but requires a few additional minutes to travel. As usual, a number of alternative routes than require less energy consumption exist, such as the one drawn in lime green, passing through Marxzell and Gaggenau.

The query from Koblenz to Limburg, shown in Figure 6.6, is a very typical one, again with the fastest connection being the motorway to the North, and a slightly slower A-road providing an alternative fast route, shown in dark red and leaving the city via the South, joining the motorway near Montabaur. A different way out of the city exists, shown in blue, as do numerous alternatives for the way from Montabaur to Limburg, e.g. the yellow one that shares its beginning with the route in blue.

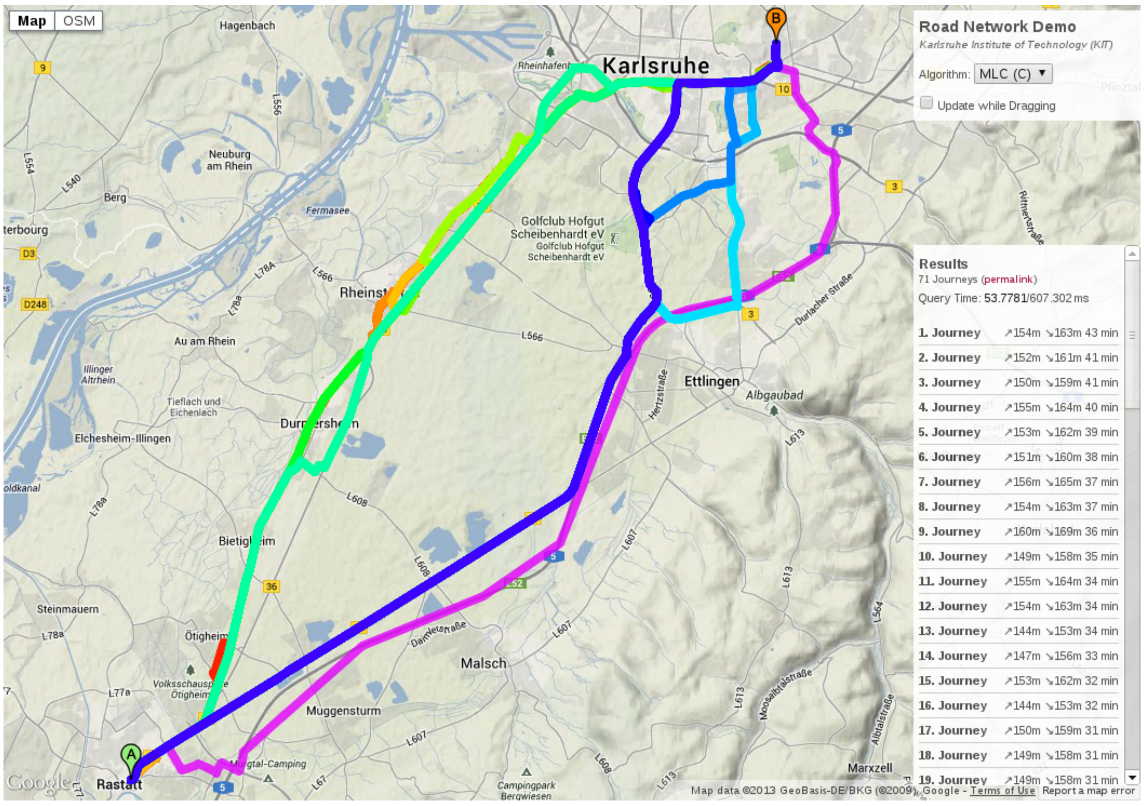
Lastly, we provide an example of a query where 16 kWh do not suffice. For the query from Karlsruhe to Ulm, shown in Figure 6.7, we set the battery capacity to 40 kWh, which comes close to the usable battery capacity of larger electric vehicles currently in production. As there do not exist many alternatives to the motorway connecting the two cities, our algorithm only finds a plethora of routes, each of which avoids the motorway on different segments. Therefore, the possible savings are mostly restricted to varying speed on the motorway. As visible in the Pareto curve shown in Figure 6.8, all Pareto points with an energy consumption above 28.3 kWh, up to the maximum of 37 kWh, or travel times from 86 to 105 minutes, belong to the 158 km long motorway route. The fastest journey on this route thus uses 30% more energy than the slowest *globally optimal* one, while it is only 18% faster. Note that the difference between the motorway routes with the purple diamond and red plus sign is the latter's usage of a slip road on a length of 1.3 km, as due to a DEM accuracy issue at the motorway junction in Wendlingen, the motorway is incorrectly assumed to have a slight slope, while the slip road does not have a vertex on the bridge, but on either side of it and thus is not affected by this glitch.


Map
OSM
Road Network Demo
Karlsruhe Institute of Technology (KIT)
Algorithm: MLC (C)
Update while Dragging
Berlin
Map data ©2013 GeoBasis-DE/BKG (©2009), Google - Terms of Use Report a map error

Results
34 Journeys (permalink)
Query Time: 3.38306/279.824 ms

1. Journey	↗124m	↘115m	35 min
2. Journey	↗125m	↘116m	34 min
3. Journey	↗125m	↘116m	32 min
4. Journey	↗137m	↘128m	30 min
5. Journey	↗131m	↘122m	29 min
6. Journey	↗132m	↘123m	28 min
7. Journey	↗126m	↘117m	27 min
8. Journey	↗131m	↘122m	27 min
9. Journey	↗127m	↘118m	26 min
10. Journey	↗128m	↘119m	26 min
11. Journey	↗133m	↘124m	25 min
12. Journey	↗135m	↘126m	25 min
13. Journey	↗134m	↘125m	25 min
14. Journey	↗136m	↘127m	25 min
15. Journey	↗145m	↘136m	25 min

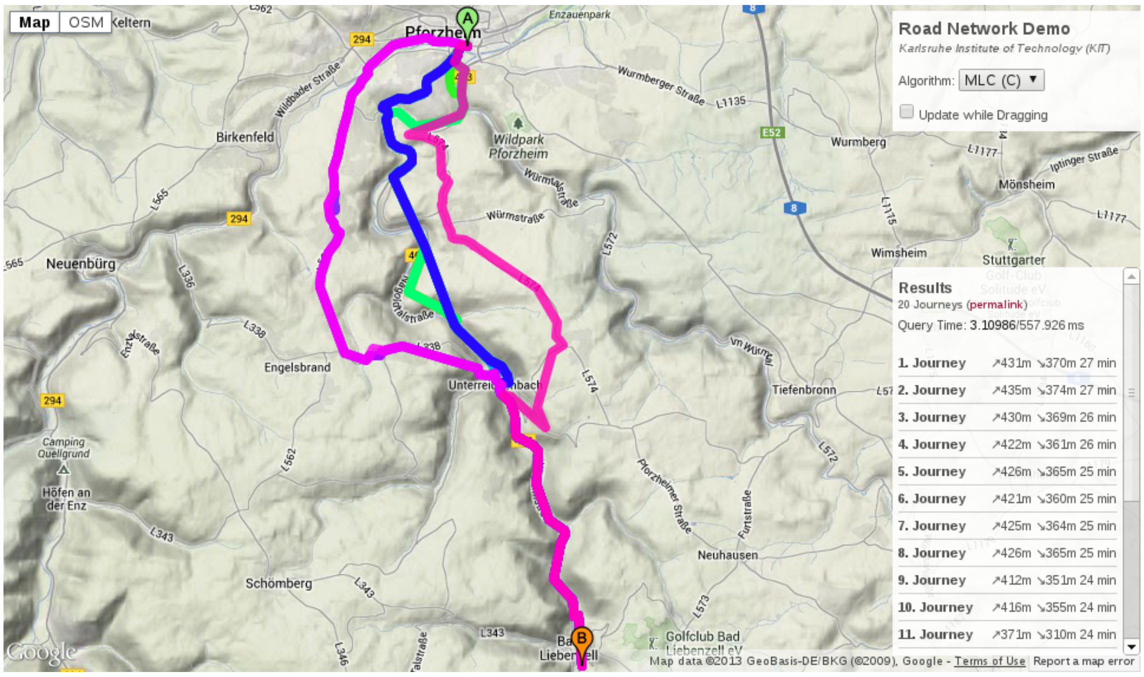
Figure 6.2: From one end of Berlin (52.4701,13.4925) to the other (52.5848,13.3378)


Map
OSM
Road Network Demo
Karlsruhe Institute of Technology (KIT)
Algorithm: MLC (C)
Update while Dragging
Karlsruhe
Rastatt
Ettlingen
Map data ©2013 GeoBasis-DE/BKG (©2009), Google - Terms of Use Report a map error

Results
71 Journeys (permalink)
Query Time: 53.7781/607.302 ms

1. Journey	↗154m	↘163m	43 min
2. Journey	↗152m	↘161m	41 min
3. Journey	↗150m	↘159m	41 min
4. Journey	↗155m	↘164m	40 min
5. Journey	↗153m	↘162m	39 min
6. Journey	↗151m	↘160m	38 min
7. Journey	↗156m	↘165m	37 min
8. Journey	↗154m	↘163m	37 min
9. Journey	↗160m	↘169m	36 min
10. Journey	↗149m	↘158m	35 min
11. Journey	↗155m	↘164m	34 min
12. Journey	↗154m	↘163m	34 min
13. Journey	↗144m	↘153m	34 min
14. Journey	↗147m	↘156m	33 min
15. Journey	↗153m	↘162m	32 min
16. Journey	↗144m	↘153m	32 min
17. Journey	↗150m	↘159m	31 min
18. Journey	↗149m	↘158m	31 min
19. Journey	↗149m	↘158m	31 min

Figure 6.3: Rastatt (48.860,8.21342) to Karlsruhe, Informatics Building (49.0135,8.41829)

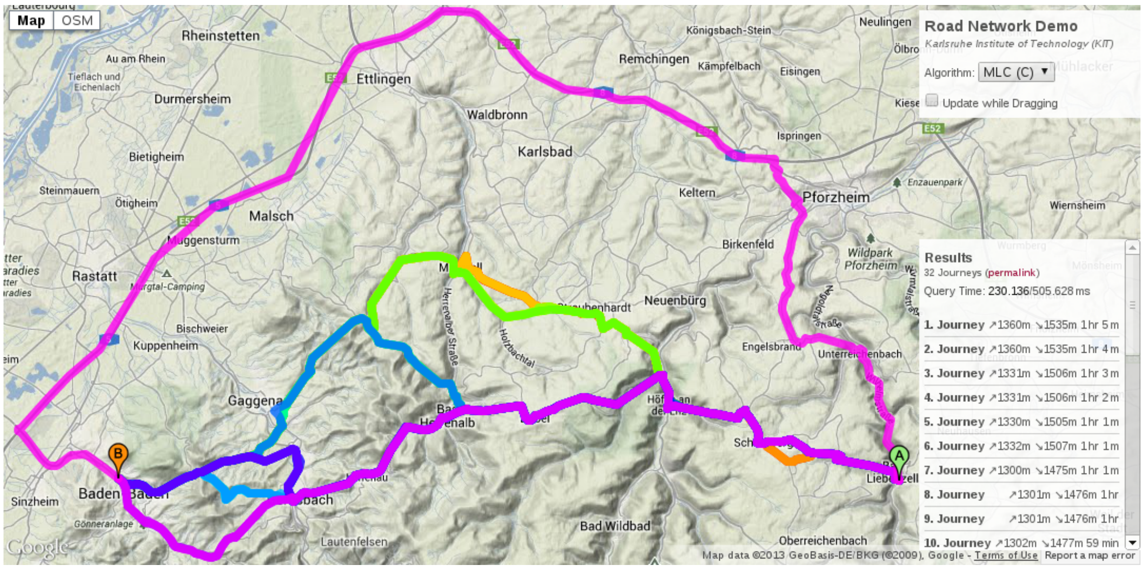


Road Network Demo
Karlsruhe Institute of Technology (KIT)
Algorithm: MLC (C) ▼
☐ Update while Dragging

Results
20 Journeys (permalink)
Query Time: 3.10986/557.926 ms

1. Journey	↗431m	↘370m	27 min
2. Journey	↗435m	↘374m	27 min
3. Journey	↗430m	↘369m	26 min
4. Journey	↗422m	↘361m	26 min
5. Journey	↗426m	↘365m	25 min
6. Journey	↗421m	↘360m	25 min
7. Journey	↗425m	↘364m	25 min
8. Journey	↗426m	↘365m	25 min
9. Journey	↗412m	↘351m	24 min
10. Journey	↗416m	↘355m	24 min
11. Journey	↗371m	↘310m	24 min

Figure 6.4: Pforzheim (48.8901,8.70043) to Bad Liebenzell (48.7714,8.73369)

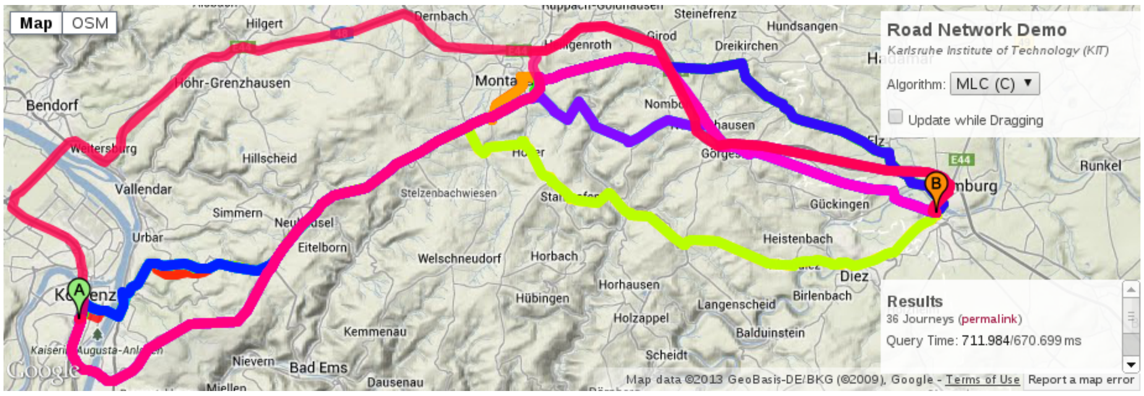


Road Network Demo
Karlsruhe Institute of Technology (KIT)
Algorithm: MLC (C) ▼
☐ Update while Dragging

Results
32 Journeys (permalink)
Query Time: 230.136/505.628 ms

1. Journey	↗1360m	↘1535m	1 hr 5 m
2. Journey	↗1360m	↘1535m	1 hr 4 m
3. Journey	↗1331m	↘1506m	1 hr 3 m
4. Journey	↗1331m	↘1506m	1 hr 2 m
5. Journey	↗1330m	↘1505m	1 hr 1 m
6. Journey	↗1332m	↘1507m	1 hr 1 m
7. Journey	↗1300m	↘1475m	1 hr 1 m
8. Journey	↗1301m	↘1476m	1 hr
9. Journey	↗1301m	↘1476m	1 hr
10. Journey	↗1302m	↘1477m	59 min

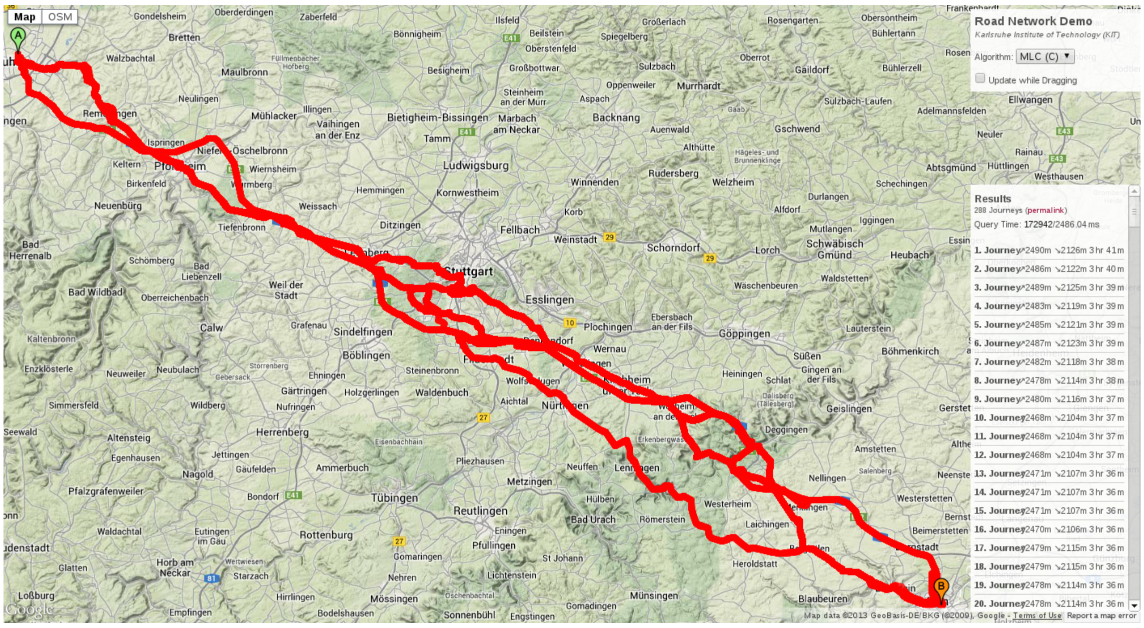
Figure 6.5: Bad Liebenzell (48.7711,8.73433) to Baden-Baden (48.7719,8.22512)


Map
OSM
Bendorf
Höhr-Grenzhausen
Vallendar
Koblenz
Montabaur
Bad Ems
Diez
Limburg
Map data ©2013 GeoBasis-DE/BKG (©2009), Google - Terms of Use Report a map error

Road Network Demo
Karlsruhe Institute of Technology (KIT)
Algorithm: MLC (C) ▼
☐ Update while Dragging

Results
36 Journeys (permalink)
Query Time: 711.984/670.699 ms

Figure 6.6: Koblenz (50.3513,7.58579) to Limburg (50.3889,8.06207)



Road Network Demo
Karlsruhe Institute of Technology (KIT)
Algorithm: MLC (C)
Update while Dragging

Results
288 Journeys (permalink)
Query Time: 172942/2486.04 ms

```
1. Journey↗2490m ↘2126m 3 hr 41 m
2. Journey↗2486m ↘2122m 3 hr 40 m
3. Journey↗2489m ↘2125m 3 hr 39 m
4. Journey↗2483m ↘2119m 3 hr 39 m
5. Journey↗2485m ↘2121m 3 hr 39 m
6. Journey↗2487m ↘2123m 3 hr 39 m
7. Journey↗2482m ↘2118m 3 hr 38 m
8. Journey↗2478m ↘2114m 3 hr 38 m
9. Journey↗2480m ↘2116m 3 hr 37 m
10. Journey↗2468m ↘2104m 3 hr 37 m
11. Journey↗2468m ↘2104m 3 hr 37 m
12. Journey↗2468m ↘2104m 3 hr 37 m
13. Journey↗2471m ↘2107m 3 hr 36 m
14. Journey↗2471m ↘2107m 3 hr 36 m
15. Journey↗2471m ↘2107m 3 hr 36 m
16. Journey↗2470m ↘2106m 3 hr 36 m
17. Journey↗2479m ↘2115m 3 hr 36 m
18. Journey↗2479m ↘2115m 3 hr 36 m
19. Journey↗2478m ↘2114m 3 hr 36 m
20. Journey↗2478m ↘2114m 3 hr 36 m
```

Map data ©2013 GeoBasis-DE/BKG (©2009), Google - Terms of Use Report a map error

Figure 6.7: Karlsruhe (49.0169,8.41784) to Ulm (48.4002,9.98479), 40 kWh battery capacity

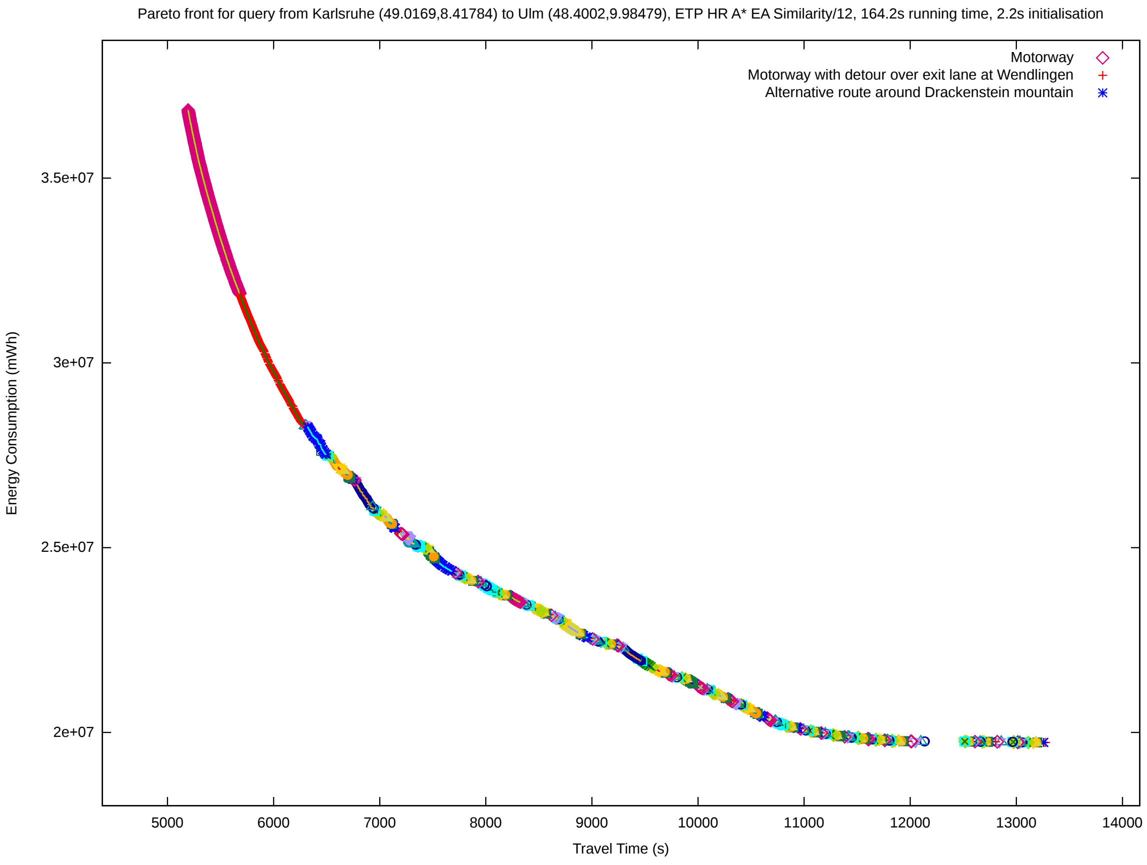


Figure 6.8: Pareto front for Query from Karlsruhe to Ulm with 40 kWh battery capacity. Key on the right refers to leftmost segments of the curve with matching symbol.

7. Conclusion

In this thesis, we developed and implemented a route-planning algorithm for battery-electric vehicles. Our method computes Pareto-optimal paths with regard to travel time and energy consumption under consideration of driving speed variation. Key challenges of electric vehicle routing include negative energy consumption through recuperation and battery capacity constraints, although here the most important issue was to minimise the number of labels, i.e. intermediate solutions, created. We demonstrated that our problem is significantly harder to solve computationally than those in route planning not considering driving speed, and exceeds the scope of practical running times even when employing state-of-the-art optimisations found in literature. To restore practicability, we developed two heuristic approaches to reduce the number of labels considered without noticeably sacrificing result quality. One of these is based on a data clustering algorithm known from literature to detect clusters of similar labels, while the other does pairwise comparisons of labels stemming from the same parent node in order to find and discard those that yield only slight improvements. In our experiments, the average running times on the road network of Western Europe, using real-world energy consumption data, dropped from several minutes to a few seconds when employing good combinations of these techniques while maintaining an average result quality of over 98% (according to our path coverage metric).

Furthermore, our experiments demonstrated the importance of lower driving speed as as alternative to resorting to other routes, a particularly compelling example of which is the query from Karlsruhe to Ulm, shown in Chapter 6.

Future Work

Several starting points for further development that exceeded the scope of this thesis remain to be examined. For one, existing speed-up techniques such as Customisable Route Planning (CRP) [DGPW11, BDP] or Contraction Hierarchies [GSSD08] could be applied to our algorithm, if an efficient approach to contraction of edges with consumption tables was developed. Such an approach could precompute the Pareto-optima for the query to be replaced by a shortcut, and apply an advanced label discarding technique to the result, which would then be the cost of the newly-added shortcut. The modelling would then have to be extended to support such edge costs, and both performance and result quality would most likely depend heavily on the label discarding procedure used.

Advanced Label Discarding Heuristics As already mentioned, label discarding techniques have a great impact on running time and result quality. Insights from research into Multi-Criterion Decision-Making and Data Mining could be used to more accurately estimate which labels might be relevant to the result, and which most likely are not. Additionally, a combination with techniques to apply fuzzy logic to route planning [DDP+13] could be fruitful. When developing such heuristics, an additional trade-off is to be made, that of running time and both the number and (ir-)relevance of the labels discarded. A procedure that accurately predicts the importance of labels to the result may therefore well take more time, as the total number of labels existing is the dominating factor of running time and such a procedure could lower that number dramatically.

Similarly, Early Aborting has potential for improvement, e.g. by measuring the difference to the closest existing label when merging a new one into a bag, and only aborting the edge scan if this difference is large enough. It remains to be seen whether the increase in result quality obtained offsets the overhead for difference calculation. Key to most of the optimisations proposed above, however, is an improved quality measure. As our current method judges routes only by path coverage and not relevance, it can only assess the similarity between path sets. While this is suitable to judge how much information is lost by a heuristic, the addition of further criteria should be considered to create a more comprehensive quality measure.

Turn Costs and Traffic Data During our experiments, we discovered that our input data does not sufficiently model the characteristics of inner-city traffic. Most notably, it lacks the constant ac- and deceleration when making turns and at traffic lights, skewing results in favour of city-centre roads. Additional experiments with penalising urban roads by assuming heavy traffic when generating consumption tables from PHEM (see Section 5.1.1) yielded promising results, but lacked sufficient data basis. Furthermore, the lack of urban traffic modelling resulted in many extremely similar routes to be found, most of which differ in negligible ways like using a street running parallel, which would no longer be viable when the energy required for making the turn were included in the computation. We therefore propose integrating turn costs into the algorithm to reduce the number of minor variations of routes, and historical traffic data to obtain a more accurate model of real-world traffic conditions. It might even be viable to compute time-dependent consumption tables, although we note that this would require modifications be made to the contraction of nodes with two neighbours and A* potential calculation. The latter could for example operate on time-independent minimum energy consumption values, or the time frame to be evaluated could be limited by forward searches on minimum and maximum travel times, similar to what we conducted for subgraph extraction.

Other Approaches Whilst our approach is designed to find *all* Pareto optima, in many cases it might be sufficient to iteratively lower a global speed limit and in each iteration choose the maximum speed possible on each road segment. This way, we would obtain several sets of paths, to the union of which a path-optimisation procedure such as the one introduced in Section 3.2 could be applied. While this approach certainly trades result quality for reduced computational complexity, it remains to be seen to which extent this is the case.

Bibliography

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

[BDP] Moritz Baum, Julian Dibbelt, and Thomas Pajor. Energy-optimal routes for electric vehicles. Unpublished.

[Bel58] Richard Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[BR11] Wolfram Burgard and Dan Roth, editors. *Proceedings of the Twenty-Fifth AAAI Conference on Artificial Intelligence.* AAAI Press, August 2011.

[DDP$^+$13] Daniel Delling, Julian Dibbelt, Thomas Pajor, Dorothea Wagner, and Renato F. Werneck. Computing Multimodal Journeys in Practice. In *Proceedings of the 12th International Symposium on Experimental Algorithms (SEA'13)*, volume 7933 of *Lecture Notes in Computer Science*, pages 260–271. Springer, 2013.

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

[DHM$^+$09] Daniel Delling, Martin Holzer, Kirill Müller, Frank Schulz, and Dorothea Wagner. High-Performance Multi-Level Routing. In Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*, pages 73–92. American Mathematical Society, 2009.

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

[DMS08] Yann Disser, Matthias Müller–Hannemann, and Mathias Schnee. Multi-Criteria Shortest Paths in Time-Dependent Train Networks. In McGeoch [McG08], pages 347–361.

[DSSW09] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering Route Planning Algorithms. In Jürgen Lerner, Dorothea Wagner, and Katharina A. Zweig, editors, *Algorithmics of Large and Complex Networks*, volume 5515 of *Lecture Notes in Computer Science*, pages 117–139. Springer, 2009.

[DW09] Daniel Delling and Dorothea Wagner. Pareto Paths with SHARC. In Jan Vahrenhold, editor, *Proceedings of the 8th International Symposium on Experimental Algorithms (SEA'09)*, volume 5526 of *Lecture Notes in Computer Science*, pages 125–136. Springer, June 2009.

[EFS11] Jochen Eisner, Stefan Funke, and Sabine Storandt. Optimal Route Planning for Electric Vehicles in Large Network. In Burgard and Roth [BR11].

[EKSX96] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In Evangelos Simoudis, Jiawei Han, and Usama M. Fayyad, editors, *KDD*, pages 226–231. AAAI Press, 1996.

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

[GH04] Andrew V. Goldberg and Chris Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. Technical Report MSR-TR-200, Microsoft Research, 2004.

[GSSD08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In McGeoch [McG08], pages 319–333.

[Han79] P. Hansen. Bricriteria Path Problems. In Günter Fandel and T. Gal, editors, *Multiple Criteria Decision Making – Theory and Application –*, pages 109–127. Springer, 1979.

[HKMS06] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors, *The Shortest Path Problem: Ninth DIMACS Implementation Challenge* -, November 2006.

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

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

[HSW08] Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering Multilevel Overlay Graphs for Shortest-Path Queries. *ACM Journal of Experimental Algorithmics*, 13(2.5):1–26, December 2008.

[HZ80] Gabriel Y Handler and Israel Zang. A dual algorithm for the constrained shortest path problem. *Networks*, 10(4):293–309, 1980.

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

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

[JP02] Sungwon Jung and Sakti Pramanik. An Efficient Path Computation Model for Hierarchically Structured Topographical Road Maps. *IEEE Transactions on Knowledge and Data Engineering*, 14(5):1029–1046, September 2002.

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

[McG08] Catherine C. McGeoch, editor. *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*. Springer, June 2008.

[MS08] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008.

[MW01] Matthias Müller–Hannemann and Karsten Weihe. Pareto Shortest Paths is Often Feasible in Practice. In *Proceedings of the 5th International Workshop on Algorithm Engineering (WAE'01)*, volume 2141 of *Lecture Notes in Computer Science*, pages 185–197. Springer, 2001.

[MZ00] Kurt Mehlhorn and Mark Ziegelmann. Resource constrained shortest paths. In *Algorithms-ESA 2000*, pages 326–337. Springer, 2000.

[SLAH11] Martin Sachenbacher, Martin Leucker, Andreas Artmeier, and Julian Haselmayr. Efficient Energy-Optimal Routing for Electric Vehicles. In Burgard and Roth [BR11].

[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*, pages 20–25. ACM, 2012.

[SWI91] Bradley S Stewart and Chelsea C White III. Multiobjective A*. *Journal of the ACM (JACM)*, 38(4):775–814, 1991.

[SWW00] Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. *ACM Journal of Experimental Algorithmics*, 5(12):1–23, 2000.

[SWZ02] Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Using Multi-Level Graphs for Timetable Information in Railway Systems. In *Proceedings of the 4th Workshop on Algorithm Engineering and Experiments (ALENEX'02)*, volume 2409 of *Lecture Notes in Computer Science*, pages 43–59. Springer, 2002.