Karlsruhe Institute of Technology

Energy-Optimal Routing with Turn Costs for Electric Vehicles

Bachelor Thesis of

Alexandru Lesi

At the Department of Informatics
Institute of Theoretical Computer Science

Reviewers:	Prof. Dr. Dorothea Wagner
	Prof. Dr. Peter Sanders
Advisor:	Moritz Baum, M.Sc.

Time Period: 1st April 2014 – 10th August 2014

Statement of Authorship

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

Karlsruhe, 10th August 2014

Abstract

This bachelor thesis focuses on the integration of turn costs in route planning for electric vehicles. First the information provided by the PHEM [HRZL09] data is used to create consumption values for turn costs. An accordingly appropriate matching principle is described to link costs of brake or acceleration maneuvers extracted from this data to turns in graphs representing road networks. Next two types of graph extensions that can hold these values are presented, along with the corresponding methods required to generate them from graphs without turn costs. Further the alterations required by the Dijkstra routing algorithm [Dij59] to process these graph types are presented. The resulting optimal path computation algorithms, along with their different graph types, are then compared to each other in a series of experiments. Lastly a case study is used to show the various differences in routing with turn costs.

Deutsche Zusammenfassung

Diese Bachelorarbeit befasst sich mit der Integration von Wendekosten in der Routenplanung für Elektrofahrzeuge. Als erstes werden die von PHEM [HRZL09] bereitgestellten Daten genutzt um Verbrauchswerte für Wendekosten zu erstellen. Ein entsprechend angemessenes Prinzip wird beschrieben, um die Kosten für Brems- und Beschleuningungsvorgänge, die aus diesen Daten gewonnen werden, auf Abbiegungen in Graphen, die Straßen Netzwerke repräsentieren, abzubilden. Danach werden zwei Arten von Graphen Erweiterungen die diese Werte speichern können vorgestellt, zusammen mit den entsprechenden Methoden die dazu benötigt werden um diese Größen für Graphen ohne Wendekosten zu generieren. Des Weiteren werden die Änderungen, die für den Dijkstra [Dij59] Algorithmus für Routenplanung erforderlich sind, um diese Graphen Typen zu bearbeiten, vorgestellt. Die resultierenden Algorithmen, zusammen mit ihren passenden Graphen Typen, werden dann in einer Reihe von Experimenten miteinander verglichen. Als letztes wird eine Fallstudie durchgeführt, die die unterschiedlichen Veränderungen von Routenplanung mit Wendekosten aufweist.

Contents

1. Introduction

Our consumer society is nearing a critical event as the fuel prices keep rising and the natural resources required for transportation rapidly near their end. At some point fuel based vehicles will inevitably be put to rest and a successor will be necessary to uphold the current standard of supply many countries have grown accustomed to. Although means of transportation such as trains and ships prove to be a more cost efficient choice for certain jobs, and some of these already run on renewable energy sources, road vehicles will never be completely replaceable. Among the alternatives to gasoline and the various other materials that currently power most of the motor vehicles worldwide electric energy is one of the few which are currently in real life use. However, electric vehicles and electric vehicle routing both still present some space for enhancement before they can effectively take over the needs and wants of users around the globe. The main limit they present is the distance which can be driven before a recharge must occur. The main course of action is to improve the battery lifetime, or optimize the estimation of the current energy level available. Depending on which routes are taken, what the weather and traffic conditions are present or what the average speed is certain destinations might be reached under the right conditions, while disregarding these aspects might lead to an empty battery just a few street corners away from the next charging station. More than anything an accurate estimate of the toll each journey will have on the battery, and whether or not these are too high, is crucial. Lots of factors have their influence on this, one of which has yet to take its place in every route planer. Most of these use the average consumption each portion of a road has to calculate the total cost of a journey while ignoring those that appear during the process of turning from one street to the next. These might not seem like much but added up they can make a significant difference. Since every intersection presents a multitude of different turns their individual toll can not be generalized. Hence a different approach is necessary to implement this feature. This thesis describes such an approach.

Related Work

The grounds on which optimal routing is conceived are tied to finding the most efficient path between two points within a graph. The commonly used solution for this problem is delivered by Dijkstra [Dij59]. Routing works differently depending on what values need to be optimised. When routing for electric vehicles the general concept changes since the shortest path is no longer decisive, but rather the one which minimizes the battery

consumption. The resulting energy-optimal routing, along with the battery constraints that are given by electric vehicles, are described by Baum et. al [BDPW13]. This type of routing can provide negative values, as demonstrated by Artmeier et. al [AHLS10], which can lead to negative cycles. Having such cycles at work in a graph keeps algorithms like Dijkstra's from terminating. Methods for finding these, or proving that none exist for a specific graph are presented by Cherkassky et. al [CGG$^+$10]. On graphs free of negative cycles virtually removing the remaining negative values can be achieved by applying potentials shifting, a technique granted by Johnson [Joh77]. While turns do not affect shortest paths they have an impact on the battery life in electric vehicles, and are hence relevant to energy-optimal routing. The concept of turns within a road graph, as well as the Customizable Route Planning speedup technique, is depicted by Delling et. al [DGPW13]. The method for adding turn costs for each intersection to a graph by storing these in different of tables is described by Geisberger et. al [GV11]. This model uses the edge-based Dijkstra algorithm, which is analysed by Volker [Vol08] and Winter [Win02]. The latter of the two previously mentioned also presents the alternative to turn cost lists, which is based on expanding the graph and adding virtual nodes and edges to accommodate turn costs.

Outline

Preliminary notions regarding graphs and routing, along with speed-up techniques, electric vehicles and a description of the source used to generate turn costs are covered by Chapter 2. Chapter 3 covers the concept of turns, turn costs and the method with which they are generated and matched to intersections. Chapter 4 begins by illustrating storage methods related to the two types of graphs with turn costs regarded in this thesis, together with the removal of negative cycles. Next it depicts the alterations to the Dijkstra optimal path algorithm to process turn cost, along with the issues that arise with them and the changes made to the corresponding speed-up techniques. Various experiments with the resulting graph types and algorithms on country scale road networks are shown in Chapter 5, and Chapter 6 presents a case study. Chapter 7 covers the conclusions drawn from this thesis as well as future work suggestions.

2. Preliminaries

This chapter covers basic concepts on which this thesis is built upon. It provides an introduction to graphs, paths and optimal path calculation, including basic notions regarding electric vehicles.

2.1 Graphs

Among a great variety of uses *graphs* can represent the basic structure of a street network. The following properties are generally required by such graphs. A graph $G = (V, E)$ is composed of a finite set of *vertices*, or *nodes* V, and one of *edges* E. Pairs of vertices $u, v \in V$ can be connected by *directed edges* $e = (u, v) \in E$. Such edges can represent one way streets, or situations where a graph G contains (u, v), but does not contain (v, u). In this case they represent intersections and streets. Two nodes u and v connected by any edge are neighbours, or *adjacent*. In order to compare routes with each other every edge e is associated with a weight $c : E \to \mathbb{R}$, resulting in *weighted edges*. Directed and weighted edges (u, v) and (v, u) can have different weights. Any two vertices are *connected* if a path between these two exists. If each vertex can be reached by all others then it is a *connected graph*. Each vertex $v \in V$ defines:

$$\text{inc}(v) = |(u|(u, v) \in E)|$$

$$\text{out}(v) = |(u|(v, u) \in E)|$$

$$\text{tot}(v) = |(u|(u, v) \in E \vee (v, u) \in E)|$$

These represent the number vertices with incident and outgoing Edges to and from v, as well as the total number of adjacent vertices. The inequality $\text{tot}(v) \leq \text{inc}(v) + \text{out}(v)$ applies. A pair of edges $(u, v), (v, w) \in E$ are *consecutive*. As described in the next section graphs must not contain *negative cycles*, even though they may contain negative weights. Additional information regarding the real life situation of the intersections and streets represented by G are required. Each node is associated with its corresponding *geographic height* and *coordinates*, and each edge records *length, road type* and *average driving speed*. The weight of an edge, as recorded in a graph, must not necessarily be its length, but it will most likely depend on it.

2.2 Paths

Within a graph $G = (V, E)$ a succession of $n \in \mathbb{N}, n \geq 2$ nodes $P = (v_1, \ldots, v_n)$ with $\forall i \in \{0, \ldots, n-1\} : (v_i, v_{i+1}) \in E$ constitutes a *path*. All such paths contain one or more *subpaths* $S = (v_i, \ldots, v_j)$ with $1 \leq i < j \leq n$. These are formally described by $S \subseteq P$. A paths that fulfills the quality $P = (v_1, v_2, \ldots, v_{n-1}v_1)$ is called *cycle*. If a weight function c is presented with the graph then the total weight of a path is defined as

$$c(P) = \sum_{i=1}^{n-1} c(v_i, v_{i+1})$$

Cycles with $c(P) < 0$ are *negative cycles*. The different paths connecting two vertices $v, w \in V$ can be compared to each other, and an *optimal path* $P^* = (v^*, \ldots, w^*)$ with $\forall P = (v, \ldots, w), P \neq P^* : c(P^*) \leq c(P)$ can be calculated. Such paths are free of any type of cycles, or *acyclic*. The weight function c may represent the length of the edges within a graph, that leads to *paths of minimum length*, or it may represent the energy cost that each edge requires in order to be traversed, resulting in *energy optimal paths*. In the latter case c can contain negative values.

2.3 Optimal Path Calculation

With all the concepts necessary to understand optimal paths covered a way to calculate them in an efficient manner is now required. To do this the probably most well known and wide spread solution for such purposes, as depicted by Algorithm 2.1, which was first described by Dijkstra in 1959, is used ([Dij59]). The Dijkstra routing algorithm requires a connected, weighted graph free of negative cycles. For a given starting vertex it calculates the optimal path and its weight to any other node. A priority queue data structure is required to keep track of the vertices currently being processed. This is a type of queue in which keys can be inserted and that always removes the minimal entry when prompted. A vertex id will also be linked to each value. Since every node must only be in the queue at most once an update function must also be available.

At the beginning the starting node $s \in V$ is picked. Then the initialization consists in creating labels for all vertices in the graph, each containing the minimal distance or weight $d(v)$, which is set to infinity, and a pointer to the previous node along the path $pred(v)$, that will be set to null. Finally s and $d(s) = 0$ are added to the queue and run the main loop.

On each iteration the loop checks for the next value in the priority queue, and stops if it is empty. If not, then it returns the vertex u with the smallest weight $d(u)$ among the ones currently processed. All its outgoing edges $(u, v) \in E$ are retrieved and the nodes which have a higher weight $d(v)$ than $d(u) + c(u, v)$ are sought. Those that are found are eligible for further processing and get added, or update their new value to the queue, as appropriate. Each one also has its predecessor set to u in order to accommodate the path output. $d(v)$ can have a different value than infinity at this point, which would just mean that a path to v had already been found, but was not optimal. After the loop halts the minimum weight and optimal path for any vertex can be retrieved. This is done by adding the chosen target node to a vector and then sequentially adding the parent of the last entry of the vector to it until this entry is the starting node. Reversing this vector provides the optimal path.

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 $\mathsf{pred}(\cdot)$

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

Stopping Criterion

Graphs that do not contain negative weights can halt at a chosen destination vertex t as soon as it is removed from the queue, having the optimal weight is given by $d(t)$. If negative weights are present a different approach is necessary. Otherwise, after reaching the destination node for the first time other nodes present in the queue might be able to reach this destination by traversing a negative edge, which can result in a lower destination distance. In this case halting after having found the optimal result with certainty can be achieved by using a *stopping criterion*.

Before running the algorithm the *path of minimum weight* is calculated. This is found by running an altered version of Dijkstra from each node within the graph separately, only traversing edges which lead to a negative weight, and storing the minimum of all these weights. Once this is done the global *minimal weight* is acquired. After this the algorithm can be launched. Every time the target vertex is reached the distance it records is compared to a global *tentative distance*, that will be initially set to infinity. When a lower distance is found, then the tentative distance takes its value and the loop goes on, running up to the point where the next value that is retrieved from the queue is higher than the sum of the current tentative distance and the minimal weight. Therefore none of the vertices remaining in the queue can reach a distance lower than the tentative distance since no further path can reduce their current one by more than the global minimum.
At this point it is safe to stop the algorithm, knowing that the optimal path and distance have been calculated for the given source and target vertices.

Vertex Potentials

The Dijkstra Algorithm in its current form will search through a graph having the weight of each path as the single relevant factor. This means that all paths in the opposite direction or up hill will be treated equally. Adding a type of preference to each node in order to minimize the search space can be realized by using *vertex potentials*, based on the descriptions of Johnson [Joh77]. Each vertex is given such a potential, which only influences its position in the queue. Hence the potential value is added to its current weight prior to adding to the queue, and removed from the value returned by the queue prior to processing. Vertex potentials have no impact on the result of the Dijkstra algorithm. Using them on graphs with negative weights removes the need of a stopping criterion.

There are various types of potentials that can be used. This thesis applies *query induced potentials*. Similar to calculating the path of minimum length, these potentials are generated by setting the initial distance for all nodes within a graph to zero, adding all of those that have negative edges to the queue and launching the algorithm. The absolute value of the resulting weight, that is each vertex's individual minimal path and initially at most zero, represents its potential and is stored. This way the vertices that have less space to be eventually reached more efficiently are processed earlier. Once the target node is removed from the queue all other weights that are still waiting to be handled, even after traversing the local minimal path that set the potential for our destination, would present a higher weight than the one currently processed. The algorithm can safely be halted at this point, and after removing the potential value the optimal path and weight that were pursued are available.
An alternative is *height induced potentials*, that favors paths down hill or at the same altitude.

Electric Vehicle Routing

Electric vehicles run, opposed to fuel vehicles, on electric energy, and not on some sort of propellant. Next to advantages like Eco-friendliness and lower travel costs per km these vehicles, or more precisely the batteries they run on, are able to regain a substantial amount of energy when the brakes are put on. This means that some maneuvers can have negative costs. According to Baum et. al [BDPW13] batteries present certain constraints. This means that there is a minimum and maximum of energy that they can store, and they will not go under or over these values. This calls for an alteration to the routing algorithm. Each search will be initiated for a certain battery capacity and current charge. Whenever an edge weight is subtracted from the current charge the resulting vertex distance is reduced to the maximum battery capacity if it rises above it and set to infinity, should it drop below zero, since it would mean that the battery was empty before the node was reached.

Figure 2.1 illustrates the result of adding or subtracting energy from the battery, depending on its current charge. M represents the upper bound of the battery capacity. Attempting to apply more costs than the current charge is not allowed, causing the function to return infinity. Charging the battery further than the capacity is also interdicted, leaving the battery at the maximum capacity instead.

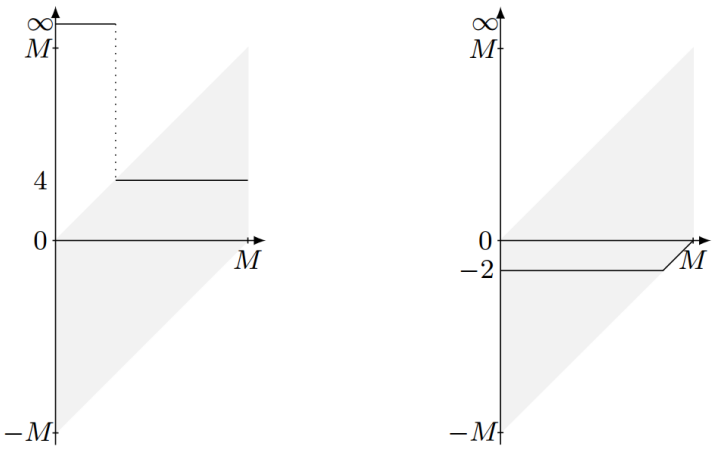

Figure 2.1: Battery constraints, taken from Baum et. al [BDPW13]

Edge-Based Dijkstra Algorithm

Certain routing approaches can cause vertices to no longer have unique parents. While this may be the case every edge handled when routing will still have its definite shortest path. As such routing is still possible, but must be done in a different manner. This effect is handled by modifying the Dijkstra Algorithm to manage edges rather than vertices, based on the work of Volker [Vol08] and Winter [Win02]. Edge labels are used, and are initialized as they would be for nodes. For a starting vertex all its outgoing edges are added to the queue, with their weights set as their current distances. When an edge is removed from the queue the node it points to is used to find further outgoing edges for processing. The increase in distance is managed the same way as before, with the difference that newly reached vertices do not have their labels edited or parents set, but the edges that point to them instead. Parent pointers also refer to the previous edge, and not vertex. In this manner, on any node, all outgoing edges can set their individual previous edge as a parent. Further alterations made to incorporate turn costs are described in Section 4.4. The stopping criterion and global minimal weight is computed and applied the same way as to the original version of the algorithm. Query induced vertex potentials however can not be used, and a variance to these, based on edges, is required. This is however not covered in this thesis.

2.4 Negative Cycle Detection

Graphs with negative cycles keep optimal path algorithms from halting, since the iteration loop will endlessly traverse such a cycle once it reaches one. This is solved with yet another alteration of Dijkstra's algorithm, provided by Cherkassky et. al [CGG$^+$10]. All vertex label distances are initially set to zero, and a search is launched from each node in turn, like when computing the query induced vertex potentials. On each run where the start vertex is pulled from the queue and presents a new distance lower than zero a negative cycle has been found. This has to be removed, and the search must be repeated for the same vertex until it terminates without finding any further negative cycles. Once all nodes have been processed the graph is prepared for use in routing. The turn cost vector graph requires an alteration to this method, which is described in Section 4.3.2.

3. Turn Cost Modelling

This chapter depicts the turn cost model. It defines turns, turn costs and describes the parameters according to which they are differentiated as well as the method used to match them.

3.1 Turns and Turn Costs

Within a graph $G = (V, E)$ or along a path $P \in G$ a *turn* $t(e_1, e_2)$ with $e_1, e_2 \in E$ presents a distinct way to traverse a vertex $v \in V$. For it to be a valid turn e_1 must be an incoming and e_2 an outgoing edge. Turns around $t(e = (u, v), f = (v, u))$ are also valid. Each vertex v presents $\text{inc}(v) * \text{out}(v)$ possible turns. This signals the fact that the total number of turns can be multiple times higher than the number of nodes within a graph.

A turn generally consists of two *maneuvers*: the first is generally a brake maneuver, that will be completed when in the center of the intersection, and an acceleration maneuver afterwards. When turning in certain situations, like up or down hill, the type of the two maneuvers may switch, change, and one or both may even become null. Two different consumption values result, one for each maneuver, and since depending on the case one of the two may be negative they cannot simply added to one single value since the battery constraints might not be respected. For example a turn $t(e = (u, v), f = (v, w))$ may have positive deceleration and negative acceleration costs. This situation will occur when a vertex is on a hill, making e an up hill and f a down hill ride. Considering a case where the deceleration costs are too high for the current battery life the vehicle will not reach the top of the hill at vertex v. Adding both costs to a single unit could cause this fact to be ignored. Hence, in order to ensure that the battery constraints are not violated, the two costs per turn must be stored separately.

Two different intersections, presenting all the possible turns that traverse them, and are highlighted by green color, are shown in Figure 3.1. The intersection on the right displays turns at different angles. These present higher to lower costs from left to right, as will be explained in the next section. A more common occurrence in road networks is represented by the other intersection, which also contains a turn around.

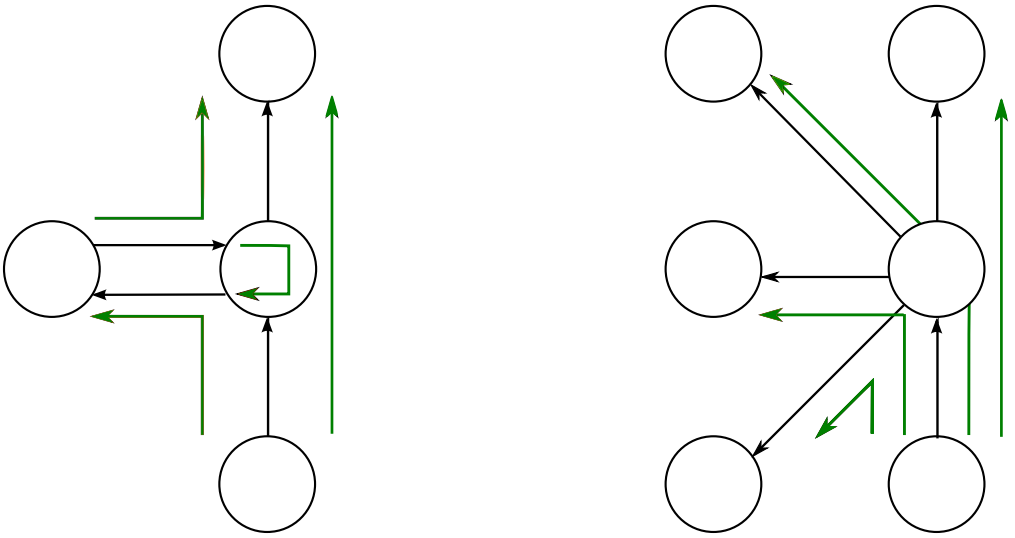

Figure 3.1: Illustration emphasizing all the turns (green) possible for two different intersections

3.2 Turn Cost Parameters and Matching

The real world street networks contain a large variety of distinct turn situations, for which different costs must be available. Each maneuver depends on certain parameters. These are given by values within the graph or gained by combining them for pairs of maneuvers that make up a turn.

Parameters

Since the two costs of a turn can not be added together the turn is split into the maneuvers that constitute it, which are then regarded separately. Each maneuver will offer a *consumption value* and *travel distance* as a result, which will depend on a combination of *start speed*, *target speed*, *slope* and *service level* (as described by Section 4.1.1). The two speed values are what the vehicle is travelling at the beginning and the end of the maneuver, the slope describes the inclination of the road and the service level the traffic situation. While it is clear that driving faster, slower, up or down hill has an impact on the consumption the latter of the four parameters remains to be justified. After observing the consumption matrix (Section 4.1.2) it became apparent that each of the service levels generally comes with its own range of allowed speeds. This occurs 'naturally', since each type of street is made to sustain a certain amount of traffic at a resulting average speed. There can not be roads with high average speeds such as $100km/h$ and stop and go traffic. However, roads with average speeds of $30km/h$ and freeflow traffic may exist. Such roads are less likely to represent the most common occurrence in a real-life street network , and saturated or heavy traffic might present a more appropriate choice. Setting the service level according to the average speed of a street ensures that the consumption values for different speeds and slopes lie within the more likely case of traffic situation. Hence adding it as a further

parameter proved to be an adequate choice.

These four parameters are gained by evaluations based on the coordinates and geographic heights of the vertices involved, the length and street type of the edges on which the turn is carried out as well as the number and road types of the edges incident to the center of the turn. The source of the resulting consumption and distance values, together with the method with which they are generated, is described in Section 4.1.2.

Matching

The matching process consists on one hand in evaluating the individual parameters for each maneuver and on the other hand in determining their common parameter, the *transition speed*.

First the difference between the heights of two nodes divided by the length of their connecting edge is calculated. This defines the angle at which the road is tilt, or the slope. Should both nodes present the same height their slope is automatically set to zero.
With the vertex coordinates composing three points in a two-dimensional plane the angle between the roads can be retrieved with one of the many formulas available, like for example the law of cosines. Regardless of the *right of way*, that will be described later on, this angle is used to calculate the transition speed. Depending on how sharp the turn is a brake maneuver or a *full stop* may or may not be required. This angle can be interpreted in a number of different manners. The interpretation used in our experiments is described in Chapter 5, together with the service level setting. After this the average edge speed is used to determine the starting and target speed for the primary and secondary maneuver respectively. The two remaining speed values are identical as they represent the transition speed over the intersection, where one maneuver ends and the other begins. Finally the right of way is estimated by comparing the road types of all the remaining edges incident to the intersection to the one that is involved in the turn. If the involved edge has the highest rank among all then the previously calculated transition speed is kept. If it is outranked by another edge a full stop is automatically selected instead.

Should there be no higher ranked edge, but n further incoming roads of the same rank, then the resulting values would be given by combining two different cases. First a brake to zero and a maneuver to the previous transition speed are calculated separately. The resulting consumptions are c_0, c_t and their distances are given by d_0, d_t respectively. The final values for this turn are then defined as:

$$c = \frac{n * c_0 + c_t}{n + 1}$$
$$d = \frac{n * d_0 + d_t}{n + 1}$$

Using these principles turn costs are matched to the majority of intersections within a graph. A few exceptions remain however. These are handled in Section 4.3.1

4. Algorithms and Data Structures

The following chapter covers the algorithms and data structures used to bring turns into routing for electric vehicles. It begins by describing the source of the consumption values, along with the method used to generate them and bringing them in a simpler form. Next the storing concepts of graphs, as well as storage methods for turn costs regarded in this thesis are depicted. Then the process of adding turn costs to graphs is shown, along with the alterations to negative cycle removal. Finally the individual adjustments of Dijkstra's algorithm on the two turn cost graph models are presented.

4.1 Turn Cost Consumption Matrix

Graphs that represent road networks contain a number of distinct turn situations. Each of their composing maneuvers calls for adequate costs, depending on the parameters described in Section 3.2. The turn cost *consumption matrix* represents the origin of consumption values used to implement turns in graphs. This section covers the source for real-world consumption values as well as the method used to process them into entries for the consumption matrix.

4.1.1 PHEM Data

PHEM is the abbreviation for Personal Car and Heavy Duty Emission Model [HRZL09]. The PHEM data offers a large number of *driving cycles*, each of which presents a simulated drive that records a great variety of information such as current battery consumption, traveled distance, speed, acceleration and current slope for every passing second. During such drives sensible break and acceleration maneuvers are undertaken. These maneuvers are generally at reasonable acceleration and deceleration paces aiming to mimic a 'normal' driving style. Driving cycles are available for most combinations of slopes, seasons, street types and service levels, the latter of which describe the traffic situation, and individually take a value among *Freeflow, Heavy, Saturated* and *Stop and Go*. While certain information further differentiates these cycles it is not regarded since the costs for a turn maneuver should either not be altered by them when the previous parameters are given or the graph can not distinguish between the new cases. For example the speed limit does not affect the consumption matrix, since the average speeds used are already within this limit. Data concerning whether or not a street is in an urban or rural area is then disregarded since no such knowledge is used within the graph used for the experiments. The information provided by these cycles presents the source of the turn cost values that are used in this thesis.

4.1.2 Generating Entries

All the driving cycles available within the PHEM data are different and each one covers a very unique situation. They are used to provide a single consumption value for every combination of the parameters presented in the previous section. The resulting consumption matrix is described by the following function:

$$M(s_1, s_2, serv, grad) = (cons, dist)$$

This function returns the costs for a maneuver starting with the speed s_1 and ending with s_2. Speeds are taken in multiples of 10 km/h and range between $[0, 15]$. $serv$ is the service level of the street, and $grad$ represents the gradient or slope of the road. The result is then a tuple consisting of the consumption $cons$ in kWs and the distance $dist$ in km that such a maneuver will require. A concrete example $M(0, 5, 2, 0)$ represents the acceleration from 0 to 50 km/h on a flat road with saturated traffic, which can generally occur in a city.

The consumption matrix is calculated for a specific season, or alternatively for a general setting without auxiliary influences. Within the selected setting all cycles are viewed separately by sequentially passing through them line by line, gathering information from each corresponding second. Each cycle is given for a certain service level, leaving only the start and target speed as well as the slope to be set. While cycles are also bound to a slope this can still vary for certain portions of the simulated drive.

Traversing through a cycle every passing second provides lots of different values, of which the total travel distance, current speed, current consumption and slope are used. Processing them is done as depicted in Algorithm 4.1. It basically works in two states on each line: resetting the current values or continuing to work with them. This reset, which is always done on the first loop, stores the read speed as s_1, the consumption as $cons$, and the distance as $dist$. $reset$ indicates whether or not one was applied in the previous loop run. Even though the driving cycle is made for a specific slope it appears this still varies between its intended value and zero throughout the file. Because of this, on each value reset, the currently read gradient is stored and the currently stored values will only be used as long as it remains the same. Additionally, if no reset was just made, the speed that is being processed must be going in the right direction, and the difference to the prior one must be larger than a certain constraint (for example 0.05). This constraint ϵ is used to keep the loop from processing bad entries that appear when the vehicle is only maintaining its current speed at the beginning or end of maneuvers. The direction dir is defined right after each reset as $\{-1, 1\}$ respectively for brake and acceleration maneuvers and used as a factor for the difference between this new speed and the previous one. When continuing to work with the current values s_2 stores the latest speed, the new consumption is added to $cons$, and the travelled distance since the last reset is saved as $dist_{out}$. Along with $serv$ both values and all four parameters required for an entry in M are now found, and so these are passed to another method. Working with the PHEM data has shown that two consecutively read speeds can present gaps of more than $10km/h$ between them. Due to this cycle portions as short as 2 lines, or seconds, are accepted. Since, at this point, it is not possible to decide if the best run was found for a certain entry the algorithm will run a method to check the found values on every non-resetting loop.
When trying to fill an entry it might already contain a value previously inserted. Due to this it is required to compare the competing values, and decide which is more appropriate. The correct spot is attributed to each speed by rounding it to the nearest multiple of $10km/h$. This means a maneuver from $14km/h$ to $16km/h$ would be associated to 1 and 2. This is a very poor result that can be ranked with the help of a function called *adequacy*:

$$adqc = \frac{(spot_{s_1} - spot_{s_2}) * 10}{s_1 - s_2}$$

Algorithm 4.1: FILL MATRIX

Input: Line of Driving Cycle, $\epsilon > 0, serv$
Output: New values for the Consumption Matrix

```
// Initialization
reset ← false
dir ← 0
grad ← 0
// Main loop
while not end of file do
    s_in ← READ()
    dist_in ← READ()
    cons_in ← READ()
    grad_in ← READ()
    if (grad = grad_in) ∧ ((reset = true) ∨ ((s_in − s_2) ∗ dir > ε)) then
        if reset = true then
            if s_1 < s_in then
                dir ← 1
            else
                dir ← −1
        reset ← false
        s_2 ← s_in
        cons ← cons + cons_in
        dist_out ← dist_in − dist
        CHECKENTRY(s_1, s_2, grad, serv, cons, dist_out)
    else
        reset ← true
        s_1 ← s_in
        grad ← grad_in
        cons ← cons_in
        dist ← dist_in
```

Only cases where $spot_{s_1} \neq spot_{s_2}$ are allowed. The optimal $adqc$ would have the value 1, while the one for the previously described case is at 5, and for $6km/h$ to $24km/h$ at ca. 0.55. This efficiently favours maneuvers that have a more appropriate difference between the two speeds, but will also regard the maneuver $14km/h$ to $24km/h$ as optimal. Limiting the sum of the absolute differences between each speed and the multiple of its spot to a small value (for example $2km/h$) solves this issue. Furthermore $adqc \in [0.9; 1.1]$ is set as a backup, to ensure that no highly inconvenient values are used to fill an entry. This however seems to not reduce the number of overall filled entries. The consumption and distance values are then multiplied with their $adqc$ before used for comparison or stored.

The new total distance of a maneuver proved to be of vital importance. Namely, if this distance is too high, several edges can have consumption values attributed to them that are not long enough to accommodate such a maneuver. No edge may have costs attributed to it for maneuvers longer than half of its length. If the shape of a graph inside a city is considered, with a high number of very short edges, long distances for a turn maneuver represent an actual problem. When filling obvious cases, where the entry has not yet been filled at all, or has a worse adequacy and higher distance, the new values are

used. The other cases seem to be handled best when taking the lowest distance within a tolerated (for example 5%) adequacy margin. Using this approach leads to satisfying results, where certain maneuvers even present perfectly identical distances for several entries. This characteristic will be utilized in a moment.

As such, after processing all the given cycles, a high number of the consumption matrix can be filled. Several entries are however still empty. This happens mainly due to the fact that many entries are either not allowed on certain service levels, or generally don't occur, like for example a brake maneuver from $140km/h$ to $130km/h$. Even though a few values that are later required can not be filled with the help of the PHEM data directly, using the ones found cover most of the real life situations, and these have turned out to be various enough to be used to estimate the missing cases. More information on this subject is given in Section 4.3.1

With the driving cycles processed the matrix still presented a few inconvenient entries, mainly due to very high distances. Several post-processing procedures were used to remove the most inconvenient of these. Visualising the matrix and manually looking for features led to realising that for the same speeds and service levels many entries have shown close to identical distances on different slopes, as mentioned before. The higher slopes however would sometimes provide far higher distance values. This occurs when a value is falsely entered by unifying two maneuvers. Such scenarios appear when only short breaks between maneuvers are undertaken, and the current speed, while still close to a certain number, constantly grows by a small amount during the pause. To remove these unwanted entries all possible combinations are traversed and the number of similar distances for the diverse slopes is counted. If more than three distances are almost the same, but not all, then entries of the ones that fall out of line are removed.

Another type of unusable values appeared for certain situations where the lowest or highest target speed would be the same for all gradients in the same service level, except for a single one that would contain a further entry. While this is a correct value it appears to be a drive simulation that has simply went a little further than expected over a rather inconveniently long period of time. Even a few extra seconds cause far higher distance values to arise, that simply can't be used for the turn cost model. These are handled by iterating over all slope combinations for specific service levels, start and target speeds and, if only a single value is found, removing it.

At this point the matrix no longer contains values with inconvenient maneuver lengths, but presents gaps in spaces where these should not arise. As described earlier these are caused by strong acceleration or brake maneuvers, where more than $10km/h$ are gained or lost in one second. The holes can be filled with the help of a spline. The one used in this work was kindly provided by Kluge[1]. Such a spline is a type of interpolation that determines the position of the missing spots by calculating the most flat function that would run through all the other points it contains. The parameters used are the consumption and the distance. For our experiments the spline is only used when it can be given at least 4 points. This particular version also linearly extrapolates the outer ends, which is used to fill another two entries in both directions, as long as these are within the accepted bounds. With this the consumption matrix is completed.

[1] http://kluge.in-chemnitz.de/opensource/spline/ as seen on the 31.07.2014

Further Attempts

Next to all the methods and functions used to create the final consumption matrix some approaches that were undertaken proved to not be acceptable. Their main objective was to fill missing entries. One idea was to accept two maneuvers that only had a short pause between them as one. While similar to what was described earlier as an unwanted result, far more cases presented themselves when actually looking for them, several of which displayed sensible entries. Unfortunately these were still inconvenient, and were sorted out by the restrictions described previously. Alternatively winning an entry by summing up two maneuvers that would compose it brought lower distances, but not sufficiently.

A further procedure that did not satisfy was using a linear interpolation method to fill in the blanks. This caused wild fluctuations between the results as the consumption values are not directly proportional to their distances, which in turn do not have the same spacings between each other. Attempting to fill all the missing entries would even cause negative distance values to appear. These methods were then ruled out and removed from usage, keeping the consumption matrix safe from new inconvenient values.

4.2 Storage Concepts

Following the calculation of turn costs the need to store these presents itself. Saving the values for all possible combinations of turns within each vertex has proved to be rather inefficient and presented the request for improvement. Two graph models that provide an alternative way to accommodate turn costs for every possible turn are presented.

4.2.1 Vertex and Edge Storage

The information regarding which nodes are connected to each other is stored with the help of an *adjacency array*. Each node will have two pointers delimiting the portion of its outgoing edges in this array. While this method is used due to its lower storage requirements directed graphs face a disadvantage. When looking at a node it can not easily be inquired which vertices have edges pointing at them. To bypass this issue all edges are added a *direction* flag and for each currently existing one a *backward edge* is annexed to the graph. Such an edge will connect the same two vertices, but in the opposite direction. The resulting *bidirected graph* can still navigate in a single direction whilst using the direction flags to avoid irrelevant edges. Graphs without this extension are referred to as *unidirected graphs*.

4.2.2 Expanded Graph

One option to store turn costs within a graph is with the help of the *expanded graph*, based on the descriptions of Winter [Win02]. As the name suggests the thought behind this approach is to add further vertices and edges to a graph $G = (V, E)$, where the edges will contain the turn costs as weights. The graph remains unidirected. Several new edges and vertices are added to maintain the battery constraints and to keep the different possible turns separate. To achieve this new components have to be added to the graph in a manner that keeps every single turn maneuver separate from another. Figure 4.1 depicts the expansion of a single vertex.

Vertex i is initially split in $\text{inc}(v) + \text{out}(v)$ new vertices. Each of these vertices is connected to a single incoming or outgoing edge. When edges in both directions $(i, j), (j, i) \in E$ are given then i presents two new nodes, ij and ji, one for each of these edges. Like this turns around can remain correct maneuvers. Next, for each turn, or for each pair of new nodes, where one connects an incoming edge and the other an outgoing one, another one is

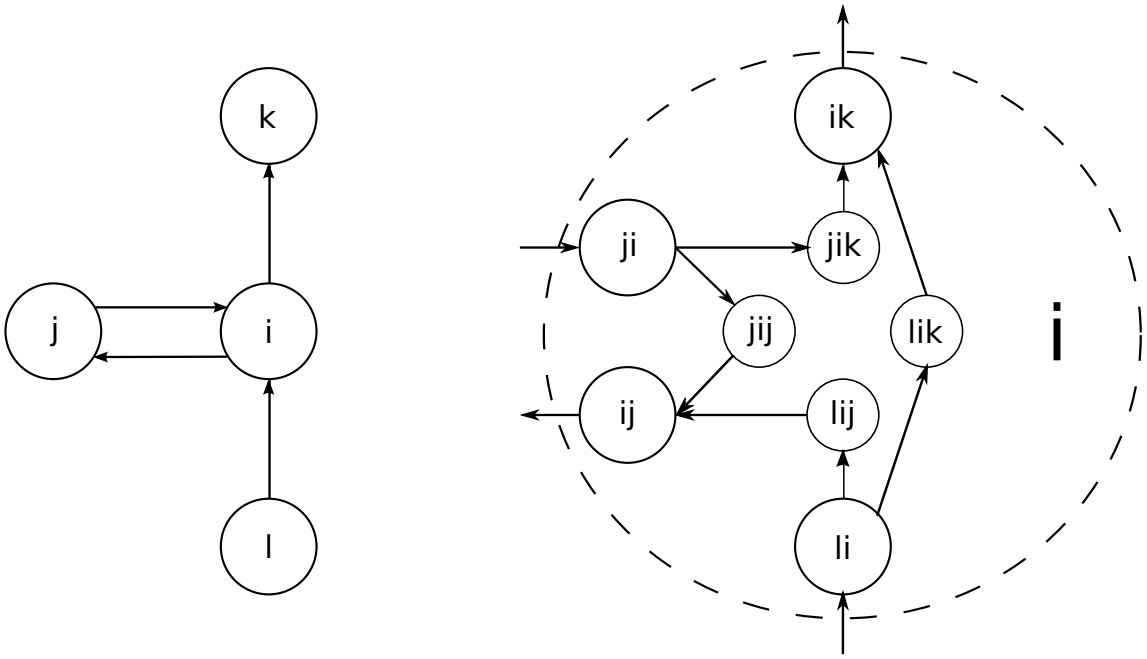

Figure 4.1: Expansion of the vertex i

created. In this manner jik is made for ji and ik. Two edges will then be added to connect this node with his 'parents', maintaining the direction of travel. The first of the described edges will have its weight set to the cost of the first maneuver for the turn in question, and the latter will present those for the second maneuver. Removing or combining any of the new vertices would either cause the battery constraints to not be respected or new routes, that should not be allowed, to become available. For instance unifying j and ij grants the option to ignore the turn costs when turning around. Removing kij and merging the two turn cost edges would cause the turn from k over i to j to disrespect the battery constraints.

When traversing the expanded graph it is not possible to differentiate between which vertices and edges where previously in the graph, and which ones have been added during the expansion. Further the graph can not directly match its current nodes to the ones in the original graph. This issue is solved by extending the graph, adding a vector that contains the pairs of nodes that connect every edge in the original version for both graph types. This way the edges that have not appeared due to the expansion can be matched to each other.

Lastly most vertices from the original graph are split into several different ones, depending on the number of outgoing edges. Due to this the majority of these new vertices can not be used as the single start for a Dijkstra optimal path search, since it will only contain one of the possibly multiple edges that would depart from the original node. Solving this requires altering the search algorithm and storing a matrix file that matches each pair of adjacent vertices from the original graph to their counterparts in the expanded one. This matrix will be refered to as the *edge association matrix*. Section 4.4 covers the corresponding Dijkstra alterations.

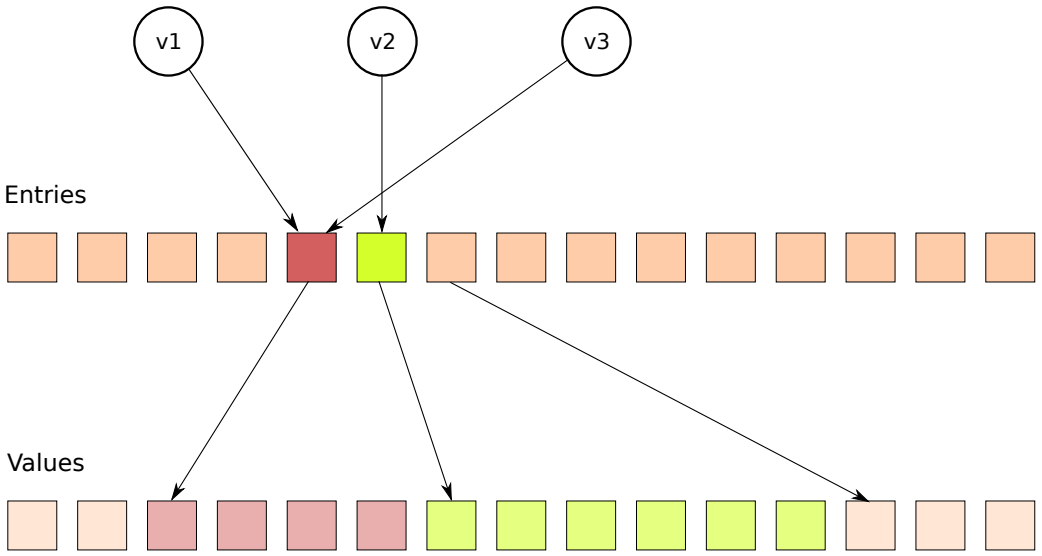

Figure 4.2: Storage of the consumption values in the turn cost vector graph.

4.2.3 Turn Cost Vector Graph

The idea of this graph $G = (V, E)$, similar to the concept of Geisberger et. al [GV11], is based on the premise that only a limited number of different intersections exist, so that several distinct ones hold an identical set of values for turn costs.

A single vector that holds the consumption values for an entire graph is used. Every distinct type of intersection will have its matching turn costs grouped together within this vector, similar to the adjacency array used for storing vertices and edges in a graph. A structure is used to define the entry for a specific intersection. It contains a pointer to the corresponding starting index in the vector of consumption values and its length, which is $2 * \text{inc}(v) * \text{out}(v)$. This length is given since every pair of incoming and outgoing edges presents two consumption values, one for the primary and one for the secondary maneuver. All vertices $v \in V$ are given a pointer to their entry, multiple of which can point at the same one, meaning that these represent identical intersections. The entry structures are also stored as a vector. Figure 4.2 illustrates this storage mechanism. The red and green entries represent intersections with one incoming edge, and two or three outgoing edges respectively.

This approach leads to a decrease in storage requirement as opposed to the expanded graph, and becomes more efficient the larger the graph due to the limited number of real-world intersection types. The result will henceforth be reffered to as a *turn cost vector graph*. With the consumption values available to all vertices within the graph the final requirement is matching these to their corresponding turns. Two index values, one for the incoming and one for the outgoing edge, must be provided by a vertex in order to retrieve the correct pair of consumption values.

The issue that presents itself is that, while the vertices of this unidirected graph can set an index for each outgoing edges with the help of their positions in the adjacency array, they can not do the same for incoming edges. This is due to the fact that the vertices in this type of graph have no direct access to their incoming edges, or the vertices that these start from. Overcoming this obstacle is done by adding all the backward edges, and making the graph bidirected. While the number of edges within the graph doubles every vertex can now access its incoming edge with the help of the adjacency array, and can set an index depending on their position. With the two indexes available the correct position for the two consumption values of each turn can be read.

Alternative Approach

As a variation a version of the graph was made where the vector of turn costs would contain every distinct consumption value only once. This meant that each vertex then contained a vector of pointers, as opposed to just one pointer, and the turn cost vector would only contain a vector of unique values, as opposed to one with every occurring value combination. Entry structures were hence no longer necessary. This attempt proved to be less efficient on graphs representing street networks the size of countries and was no longer tended to.

4.3 Graph Extension

This section covers the manner in which a graph is extended by turn costs taken from the consumption matrix. The challenges that present themselves in this process are described, along with the individual approach used to edit the two different types of turn cost graphs. Additionally the adjusted cycle removal method applied to each graph after extension is presented.

4.3.1 Integrating Turn Costs

Routing with turn costs requires the graph in use to contain consumption values for every possible turn. These are acquired by iterating over every vertex within a graph, evaluating all turn situations, and annexing the consumption values.

For the two maneuvers involved in a turn each is tied to the edge it is executed on. Since the consumption value of such a maneuver also covers the energy required to travel its distance it can not be processed as is, since it would cause the normal traversing costs for the maneuver distance to be regarded twice. This is dealt with by subtracting a reduced part of the weight of the edge in question, depending on the ratio between the length of the maneuver to that of the edge. The new resulting values are used as the final version of turn costs when routing, and can now be added to the graph.

This approach leads to a minor number of routes being omitted. These appear when the vehicle does not have enough battery capacity left to fully traverse an edge, but would be able to reach it if the costs for a brake maneuver are regarded earlier. Removing this issue can be achieved by reducing the weight of edges, and not subtracting their value from turn costs. However, since every turn maneuver presents its own length, this reduction can not be done universally. Alternatively each edge would require to have its weight reduced to accommodate the turns with the longest distances in both directions, and four turn costs must be processed per turn, the two new ones simply being the corresponding weight of the edges that was reduced more than necessary for each specific turn. This approach is not dealt with in this thesis, making the turn cost routing model currently regarded a choice that heuristically neglects very few situations for the added benefit of halving the number of turn cost values stored within an according graph.

Turn Cost Vector Graph

Each intersection is processed, turn by turn. Incoming edges are paired with outgoing ones in the order in which these are stored on the adjacency array for the current vertex, and the consumption values retrieved for each case are stored in a vector. This order is maintained for the pairs of turn costs, making it possible to inquire the correct entry when routing. The resulting vector must then be annexed to the graph.

The basic approach lies in adding a new entry by traversing all the current ones within the graph, checking for an identical one. Should such an entry be found then its index is retrieved. If not then the new unique entry is added to the back of the vector, gaining its corresponding entry structure. However, when routing on road networks the size of countries, quite a few different types of intersections present themselves, leading to a large vector, that would need to be traversed after each vertex evaluation. This method proves to be naive and presents an inconveniently high run-time, even in preprocessing circumstances. As an alternative the graph is created without comparing entries, and compressed once all nodes have been handled. This is achieved by creating entirely new vectors for entry structures and consumption values. The initial entry vector is traversed sequentially, and every entry that is found is compared all the entries further back in the vector, setting them to null should they be identical. The part of the consumption vector as well as the corresponding entry structure are then copied to the new vectors. Each vertex has its entry index updated accordingly throughout the process. With this the turn cost vector graph is complete.

Expanded Graph

Considering that the graph type regarded in this work uses an adjacency array to store edges actually expanding it is rather inconvenient. A more elegant procedure is to create an entirely new graph from the bidirected version of the source graph. Sequentially, each vertex is split in the manner described earlier, all incoming and outgoing edges are used to fill the edge association matrix, as well as to provide turn costs, that are then stored as new edges connecting the nodes that have been currently created. When viewing a vertex that has an outgoing edge pointing to one with a lower id the association matrix is used to indicate which new pair of vertices this edge must be copied to. After iterating over all nodes within the original graph the expanded version is complete, together with its association matrix.

Inappropriate and Missing Matches

The matching principles described in Section 3.2 cover the majority of possible turns for graphs based on road networks. A few exceptions arise however. These appear in two cases: when the entry itself in the matrix has not been filled and when the distance of the edge is too short to accommodate the maneuver that has been matched to it.

Missing matrix entries appear when particular situations arise, which begin and end at odd speed choices. The most frequent case is for maneuvers with a minimal change in speed, like for instance a turn from a street with $30km/h$ average speed to one with $40km/h$. These situations are handled by raising *serv* to the next level, like from *Stop and Go* to *Heavy* in this particular case. The remaining blanks appear in the opposite case, where high differences between the two speed values exist, like for example when mounting a highway. Raising the transition speed to a more realistic value, bit by bit up to the arithmetic mean of the two speeds, until an existing entry is found in the consumption matrix solves this issue.

Maneuvers with distances higher than half of an edge's length appear when attempting to do a full stop in situations where this is not entirely possible. Similarly to the previous description the transition speed is raised bit by bit, up to the according mean, and the first result found is taken. In very few cases this does not provide a result. Then the only remaining course of action is to find the transition speed with the minimal length difference over the accepted bound for both edges involved in a turn, and scaling the consumption value down, according to the ration between half of the edges length and its corresponding maneuver distance.

4.3.2 Negative Cycle Removal

Adding turn costs to a graph which contains weights that are not naive and already account for turn penalties to some extent can cause a small number of negative cycles to arise. Should these appear they have to be removed in order to ensure that the routing algorithm will always terminate, and for that need to be found first. Running an altered version of Dijkstra's algorithm, as described in Section 2.4 helps find these cycles. The alterations are applied to the versions of the routing algorithm described in the next chapter, depending on the graph type in use.

Once a cycle has been found the absolute value of the resulting negative weight is divided by the total number of traversed vertices in the expanded model, and by three times the number in the turn cost vector model. When the two turn cost graphs are based on the same initial graph then these values will be identical for the same cycles. The cycle is removed by adding this amount, plus a small further penalty (for example $5mWh$), to the weight of every edge along it, as well as updating the according pair of consumption costs for each involved turn. After this, when traversing the cycle, a distance barely above zero will be reached, and no negative value. After handling a cycle the removal algorithm can be commenced until all vertices have been treated, leading to graphs with turn costs free of negative cycles.

4.4 Dijkstra Adjustment

Altering the graphs to accommodate turn costs has an impact on the routing algorithm. Each of the graph variations call for their own adjustment, regarding the input and halting method as well as the loop iterations. The resulting mechanism handles turn cost on every turn throughout the graph, with the premise that the vehicle travel at the corresponding average speeds when starting and finishing the traversal of a path.

Expanded Graph

Routing on the expanded graph is bound to its corresponding association matrix. Only vertices presented as starting points of edges that also exist in the original version of the graph are allowed as starting points. Consequently, only those in which such edges end may be set as destinations. Otherwise it would be possible for routing to begin or end in the middle of turn maneuvers. Furthermore, when initializing the queue by adding an origin vertex, all the vertices that have been created from the same node, representing its connection to outgoing edges, must also be added to the queue. Each of these vertices o has given $d(o) = 0$ set, after which the iteration may begin. In the same sense a group of nodes that connect the incident edges of a node in the original graph are all valid destinations. Whenever one of these is reached the tentative destination distance is updated. After the routing algorithm terminates the optimal path is retrieved by starting the reverse search from the destination vertex with the lowest distance, and halted once one of the source vertices is reached. As such routing can correctly be undertaken on the expanded model.

Calculating the query induced potentials is done in the same manner as on the original graph. The stopping criterion is run in the same way as well, with the restriction that only vertices that represent connections to outgoing edges in the original graph are added to the query prior to the search, and the lowest distance is taken from one of the nodes that connects an incoming edge from the graph without turn costs.

Turn Cost Vector Graph

When routing on the turn costs vector graph vertices no longer have unique parents. This is due to the cost of traversing an edge onto the next node depending on which previous edge was traversed. All outgoing edges of a vertex can hence have individual parent edges. This effect is handled by using the edge-based version of the Dijkstra algorithm. As such the correct path can always be retrieved after the algorithm terminates.

The turn cost vector graph must now be altered to process the turn costs on each loop. When an edge is extracted from the queue it is used to retrieve the node it points to. As such, when iterating over all the outgoing edges of this vertex, a second edge to pair the one retrieved from the queue is available, and the necessary turn costs can not be looked up. Calculating the new distance is done by adding the costs of the first maneuver, then the second, and finally the edge weight to the distance that was retrieved from the queue, each processed separately, in order to not violate the battery constraints. This way routing on the turn cost vector graph is achieved and presents for matching sources and destinations the same resulting distance and paths as the expanded graph. While the stopping criterion is calculated in the same manner as described earlier the query induced vertex potentials require an alteration that has not been covered in this thesis.

5. Experiments

The following chapter covers the experimental part of this thesis. It begins by describing the actual changes that are made to the graphs in order to accommodate turn costs. Next it presents the general impact of turn costs on routing with the Dijkstra algorithm. Finally the average results for routing on multiple queries are provided. These query experiments are aimed at evaluating the storage and run-time increase caused by processing turn costs in routing.

The server used in these experiments, as well as for preprocessing purposes, runs on Intel Xeon E5430 with a total of 8 cores, clocked at 2.66GHz, and with 32GiB of RAM. A single core was used for all queries. The server runs on openSUSE 12.2, and the compiler used is GCC version 4.7.1, with -O3 optimizations. The road network used was that of Germany, which was provided by the PTV AG. Attributing weights to the edges within the graph is done based on the information given by the PHEM data, as described by Baum et. al [BDPW13]. Height information for the vertices stored within the graph is obtained from the freely available NASA Shuttle Radar Topography Mission[1] data.

Various Settings

The interpretation of the angle between two streets involved in a turn and the definition of the service level can be done in various ways. In our experiments *Freeflow* applies to speeds of $100km/h$ and above, which generally represent highways. Then *Saturated* ranges down to $51km/h$, depicting most of the remaining traffic situation outside of cities and towns, and some urban cases. *Heavy* is used for the remaining speeds above $30km/h$, which apply to general traffic inside towns and cities. Lastly *Stop and Go* is used for the leftover streets with very low average speeds. For any turn, should the road only curve less than 30 *degrees* in either direction, the lower of the average speeds from the edges involved in it is set as the transition speed. Otherwise, as long as the road does not curve more than 90 *degrees* a fixed value for traversing turns is used, 30 *km/h* in our case. All turns on roads that curve more than 90 *degrees* have a transition speed of 0 *km/h*, and represent a full stop. This matching method resulted in a total of 11% full stop situations on the entire graph of Germany.

[1] http://www2.jpl.nasa.gov/srtm/

Storage

Adding the turn costs to a graph by either expanding it or storing these as a turn cost vector with respective vertex pointers provides the first analytic results. As seen in Table 5.1 the new storage size of the routing graph for Germany are less than triple for the turn cost vector model, but more than nine fold for the expanded version. While the turn cost vector graph contains the same amount of vertices and just double the number of edges the major part of the size increase is tied to the actual vector of turn costs and the extra pointers in each vertex. Among the $4,692,091$ vertices there are only $2,947,002$ unique intersections, leading to a vector compression of over 37%. The expanded graph stores all the turn values on new edges, connecting new vertices that virtually split up every intersection. These are matched to the original vertices with help of the association vector, that makes up around 11% of the graph's size, roughly the size of the original graph. While this is a considerable amount of storage space the main size increase of the graph is solely due to it containing more than ten times the vertices and six times the edges, compared to before the expansion.

	Original	Turn Cost Vector	Expanded
Size (MB)	349.9	808.8	3,330
# Vertices	4,692,091	4,692,091	50,144,841
# Edges	10,805,429	21,610,858	67,873,395

Table 5.1: Size values regarding the different versions of the Germany graph.

Distance Increase

The following test results will be presented for the original and expanded graphs. The actual routes and necessary battery energy levels resulting from specific queries will be identical to the results of routing on the turn cost vector graph. All of the routing experiments have been performed on 1000 independent queries with randomly selected origin and destination vertices. Each presented a restricted battery capacity: either 16, 85 or 1000 kWh, and were started at full charge. The third range used for the queries is far higher than the necessity for any optimal route withing Germany and is mainly chosen to ensure a group of queries where absolutely every two distinct pair of vertices can reach each other.

The only prerequisite of these queries was that the destination vertex always lie within the chosen capacity for both graph types. Such pairs of vertices can be found by running a Dijkstra search on one of the graphs, without using any stopping criterion or potentials, from an arbitrary source vertex. After the search has finished a vertex is picked uniformly at random and, should this not coincidentally be the origin vertex, it is accepted as the destination vertex if it has been reached in the previous Dijkstra run and can also be reached by a search on the other graph type. If this is not the case then the process is repeated until a fitting pick is found. Since the expanded graph generally presents higher route distances due to turn cost penalties it has been used as the primary search graph. At first all three capacities were used to run queries on both graphs with no stopping criterion and using dummy potentials. Since the same queries of each capacity will be used for the experiments using stopping criterion and query induced potentials the resulting average consumption differences remain identical.

As seen in Table 5.2 the turn cost model has a stable increase in the average energy requirements depending on the battery life. The test with the longest range has a similar rise to the next lower one due to the limited size of the graph, meaning that only a part of the maximum energy available is required to reach any vertex from any other vertex. The higher value was expected since the consumption values of all edges are based on their average speeds, which already regard some penalties for turns. To reduce the variance to null these penalties would have to be removed from every edge.

Query Range	Original	Turn Cost	Increase
16 kWh	9,830 kWh	10,277 kWh	4.55%
85 kWh	42,868 kWh	49,288 kWh	14.98%
1000 kWh	54,972 kWh	63,352 kWh	15.24%

Table 5.2: Increase in the average optimal distance in.

Performance

The following tables provide information regarding the average results of running 1000 queries for the three different ranges on each of the graph types containing turn costs, as well as the original model. Table 5.3 represents the experiments that are run without the use of any speed-up techniques. This means that for every query the full range of the battery capacity was traversed in every possible direction before halting. For the highest range the average run-time on the original graph is slightly less than three times lower compared to the turn cost vector model (TCV), and around six times lower than that of the expanded graph. This average time also grows faster, the higher the battery capacity, on the graphs with turn costs, compared to that without. While queries on the expanded graph processes far more vertices and edges than the other two those on the turn cost vector model actually regards less edges than the original one on average for the shortest range. This advantage however is neutralized, the higher the range, and as compensation routing on the turn cost vector graph scans far more vertices. This result occurs due to the fact that routing on this model runs on the edge-based version of Dijkstra's algorithm. Hence edges are pulled from the query, and not vertices, due to which comparing these values can not quite give a definite insight.

	Original	TCV	Expanded
Time (ms)			
16 kWh	74.690	171.800	325.255
85 kWh	955.843	2,654.550	5,035.560
1000 kWh	1,128.070	3,593.050	6,815.000
Vertiex scans			
16 kWh	339,368	1,686,786	2,969,802
85 kWh	4,106,189	23,335,384	41,151,430
1000 kWh	4,853,582	31,628,568	55,800,160
Edge scans			
16 kWh	784,709	636,589	4,016,594
85 kWh	9,488,365	8,817,054	55,644,841
1000 kWh	11,209,766	11,958,900	75,443,406

Table 5.3: Average results for queries without speed-up techniques.

Table 5.4 depicts the results for the same queries on the given ranges, with the added feature that the stopping criterion was used. The average run-time is roughly halved on all three graph types. As the advantage in average edge scans is lost on the turn cost vector model it still remains relatively close to the result provided by the queries on the original graph. Overall around half of the scans are done for both edges and vertices on all three graph types, indicating that the stopping criterion provides a similar amount of speed increase among the three different types of graphs.

	Original	TCV	Expanded
Time (ms)			
16 kWh	27.880	86.527	160.801
85 kWh	410.625	1,329.130	2,448.658
1000 kWh	553.692	1,780.910	3,315.150
Vertex scans			
16 kWh	129,275	866,370	1,529,208
85 kWh	1,770,085	11,621,239	20,491,263
1000 kWh	2,392,027	15,718,746	27,720,759
Edge scans			
16 kWh	298,532	327,410	2,067,631
85 kWh	4,093,865	4,388,167	27,712,343
1000 kWh	5,530,565	5,937,293	37,486,692

Table 5.4: Average results for queries using the stopping criterion.

Query induced vertex potentials provide the method used in Table 5.5 to reduce the average duration of the various queries. As the turn cost vector graph requires an alternate version of these potentials it is not part of this experiment. The potential computation requires 5841.06 *ms* on the original graph and 36820.40 *ms* on the expanded one. Compared to the stopping criterion a further 10% increase in speed is acquired by the queries on the original model, and 24 % by those on the expanded one. This slightly more significant increase in speed, along with a more relevant drop in vertex and edge scans, as compared to routing on the original graph arises due to the fact that query induced potentials have a more compelling effect the higher the number of vertices and edges within a graph. The ratio between the vertex scans on the two graph models is nearing the one that is given by the total number of vertices within the entire graph. As this type of potentials presents a benefit to the expanded graph it does not necessarily do the same for the turn cost vector graph (when altered to the appropriate form). Another possible cause for the drop in average run-time relates to the fact that the nodes that represent parts of turns are also granted potentials. This leads to the fact that not all turns must be processed for a certain vertex immediately when routing, leaving the turns with less convenient potentials for later processing.

	Original	Expanded
Time (ms)		
16 kWh	25.841	135.088
85 kWh	373.495	1,982.220
1000 kWh	502.953	2,674.530
Vertex scans		
16 kWh	126,303	1,392,536
85 kWh	1,709,776	18,336,143
1000 kWh	2,310,138	24,793,938
Edge scans		
16 kWh	290,390	1,885,048
85 kWh	3,940,798	24,825,530
1000 kWh	5,323,375	33,566,312

Table 5.5: Average results for queries using query induced vertex potentials.

In summary turn costs require extra storage, and the turn cost vector model best compresses these. While turn maneuver values do not contain the cost for travelling the distance they are performed on a slight increase in average total consumption is present, and increases with the length of the route. Queries run on both turn cost model approaches longer and scan more vertices and edges than those on the original graph. Using a stopping criterion presents a uniform benefit on all graphs types. Expanded graph queries gain a higher benefit form query induced vertex potentials due to the higher number of nodes and possibility to add potentials to specific turns. The actual effect of the selected settings for turn cost matching is described in the following chapter.

6. Case Study

Efficient routes delivered by the turn cost model will, in most cases, be different to those from the original one. There can only be speculations to the concrete change in behaviour without visualising the different routes. Certain characteristics of these changes have become apparent in the following case study.

The figures presented in the final part of this chapter represent optimal routes for the same source and destination vertices within Germany. Yellow routes represent the optimal results that are provided by the Dijkstra Algorithm on both graphs. The orange and green paths that split from the yellow ones present the alternative subpaths that are calculated by the same algorithm on the original and expanded graph respectively. Source vertices are denoted with an S, and target ones with a T. Figures that contain only a portion of the route have T indicate the direction of the target.
While observing a larger number of routes on rural areas it directly becomes apparent that while highways or other forms of roads with high average speeds of travel are generally the quickest they do not necessarily represent the most energy efficient choice. The obvious cause is that higher speeds commonly imply higher energy consumption. Driving on a street with less speed than the average was not regarded as an option since this would only be realistic on a traffic-free setting, and can otherwise cause dangerous situations. A result of this premise is that energy optimal routes will often favour roads with lower average speeds than highways or similar types of streets. This applies to efficient routes with turn costs as well, but does not imply that the two models follow the same paths between different towns.
As seen in Figure 6.1 both graph types can provide roughly the same route even on fairly long distances. Figure 6.2 on the other hand presents a case where the two routes split away from each other for their largest parts. A closer look at them revealed that, given the option, the turn cost model will sacrifice a few extra kilometers to traverse a path that contains as few inconvenient turn situations as possible. Such situations range among towns that do not provide the option to traverse on the outskirts or intersections with other major roads. The turn cost model generally prefers the routes that contain a lower number of intersections, or with the main part of the intersections being with roads of lower rank than the one currently being traversed.

Figure 6.1: From the borders between Germany, Poland and the Czech Republic towards the north end of Germany. (50.956389, 14.722592) to (53.775122, 11.285350)

Figure 6.2: Münster to Obersulm. (49.496989, 11.755372) to (49.132533, 9.357025)

Figure 6.3: From east of Nuremberg towards the north side of town.
(49.475575, 11.071019) to (49.475575, 11.071019)

In Figure 6.3 the route provided by the original model traverses the path that would be the general choice when entering the city of Nuremberg. This however is an urban area and while the most intersections encountered will be with roads with lower rank some will always be of equal, or in some cases even higher rank. When considering the turn costs the optimal alternative is to leave the urban roads and travel on the outskirts of town as far as it makes sense. The turn cost model's result splits away from the city street and mounts the national highway *B14*, that has the clear advantage considering the situation of the intersections and is hence picked for the most energy efficient route with turn costs.
A road with the same rank as the ones on the intersections encountered along the path does not present any special advantages, and if this road should enter an intersection with a street of greater rank then it will generally be at a disadvantage. Neither of these cases presents a favorable choice when regarding turn costs. Hence, for traversing a moderately large city, the preference of the turn cost model lies in following the road that outranks the others in terms of street type. Should this not be an option the next best choice is to minimize the number of encounter with higher rank roads and take paths that intersect with streets of at least the same rank. Unlike smaller towns cities generally present a few alternative routes, making the choice possible.
In Figure 6.4 the Dijkstra Algorithm run on the original graph advises to leave a higher rank road and follow an urban path that ends back in the same road. The advantage lies in evading the intersection with a highway. While some energy might be spared this way the route does not account for the intersection with and turn to another road of the same rank or the full stop necessary when rejoining the road of greater rank. Using the expanded graph to calculate the same route deems the intersection with the highway less energy consuming and advises to not leave the route and traverse the urban area.

Figure 6.4: Passing through Karlsruhe. (48.957697, 8.475514) to (49.086464, 8.396597)

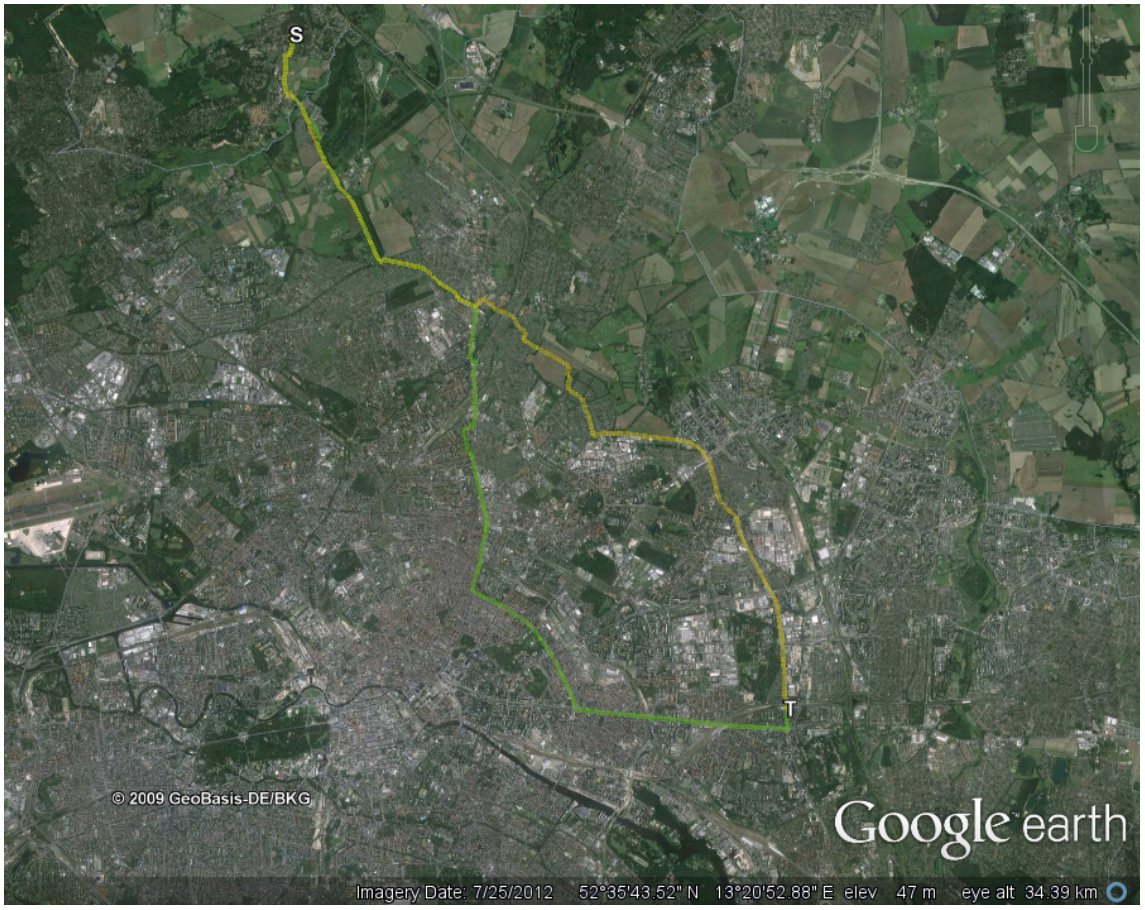

Figure 6.5: From north of Berlin to the east side of town.
(52.647803, 13.380408) to (52.513492, 13.520236)

The final case has shown that interesting path choices are tied to routing on very large cities. Taking a city the size of a capital it becomes obvious that the highways or national highways are the optimal way to get from one side to the other. However, when the destination of the route is somewhere within the city, different options arise depending on the case. In the example of Figure 6.5 the path enters the capital of Germany from the north and tries to reach its destination on the east side. Since Berlin is very large travelling on the outskirts in the rural area is not an option for an energy efficient route. The path given by the original model is fairly direct and goes through the urban part of the city, encountering a high number of intersections with other urban streets of the same rank. This results in multiple turn cost penalties, that are not calculated when using the original graph. Applying the Dijkstra Algorithm on the expanded graph reveals a route more considerate to the battery life that advises to mount the highways and use these to reach the destination, avoiding the penalties that would have applied when traversing the other route.

As a summary the turn cost model, when possible, returns routes with fewer intersections in total, or with less that are inconvenient, at the expense of slightly higher travel distances. The alternative with the fewest meetings with other roads of the same rank or higher are preferred. Longer travels avoid as many towns as possible, and short travels within a city avoid urban areas when possible. These routes can vary from the results given by the original model, although this must not occur as a rule.

7. Conclusion

In this thesis different methods to implement turn costs into routing for electric vehicles are viewed. It consists in gaining consumption values for turn costs, adding these to a manner of graph that can accommodate them, and finding compatible routing algorithms and speed-up techniques.
The PHEM data proved to present a feasible source of information for consumption values used in various turn maneuvers. While most of the maneuvers existing in road networks are covered some of these were not represented within this data. Differentiating maneuver costs by regarding speeds, slopes and service levels provided satisfying results, as shown in the case study. While some negative cycles arise due to this approach they constitute only a small number of situations.
Expanding the graph, or storing consumption values as a vector both consist in valid approaches to routing with turn costs. The expanded graph can be regarded as a rather naive approach, and provides less advantages than the turn cost vector graph. This applies to the storage situation as well as the results obtained when routing. The edge-based version of Dijkstra's algorithm is necessary to maintain a correct result on the turn cost vector model. Using a stopping criterion brings forth very similar benefits to graphs with turn costs and to graphs without these. Finally, query induced vertex potentials present a higher advantage to outing on the expanded graph as opposed to the original one.

Outlook

This thesis presents several directions in which further work can be undertaken. Attempts can be made to win even more consumption values from the PHEM data, and different heuristics can be used to select an appropriate result among the multiple ones available for the same situations. Further parameters can be matched to turns, and the service level can be given a different impact. The turn cost vector graph still contains a number of identical intersections, that are not regarded as equal due to a different edge indexing. Reordering the edges in the graph can result in a further compression of the number of entries that appear in this type of graph when creating it.
The query induced vertex potentials can be modified to work with the edge-based Dijkstra algorithm. Other types of vertex potentials can be altered and applied to routing with turn costs, as well as compared to each other. Finally further speed-up techniques, like Customizable Route Planning [DGPW13], or even Contraction Hierarchies [GSSD08], can be adapted to work with the turn cost graph models, and the result can be tested, to find if routing with turn costs can be brought to a form where it is efficient enough to take its place in all types of route planning software and hardware.

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.

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

[CGG+10] Boris V. Cherkassky, Loukas Georgiadis, Andrew V. Goldberg, Robert E. Tarjan, and Renato F. Werneck. Shortest-path feasibility algorithms: An experimental evaluation. *J. Exp. Algorithmics*, 14:7:2.7–7:2.37, January 2010.

[DGPW13] Daniel Delling, Andrew V. Goldberg, Thomas Pajor, and Renato F. Werneck. Customizable Route Planning in Road Networks. Submitted for publication, 2013.

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

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

[GV11] Robert Geisberger and Christian Vetter. Efficient Routing in Road Networks with Turn Costs. 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 100–111. Springer, 2011.

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

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

[Vol08] Lars Volker. Route Planning in Road Networks with Turn Costs, 2008. Student Research Project. `http://algo2.iti.uni-karlsruhe.de/documents/routeplanning/volker_sa.pdf`.

[Win02] Stephan Winter. Modeling Costs of Turns in Route Planning. *GeoInformatica*, 6(4):345–361, 2002.